Hardware-optimized, Symmetric, High-Performance Auto-Parallelization System with Loop-Level Parallelization, and Method Thereof

Field of the Invention

The current invention generally relates to multi-processor systems, in particular multi-core systems and parallel computing systems allowing multiprocessing in which two or more processors or cores work together to execute multiple program codes and/or processor instruction sets simultaneously, the multi-processor systems or parallel-computing systems comprising a number of central processing units linked together to enable said multiprocessing or parallel processing to take place. In particular, it relates to compiler systems for optimizing a computer program for execution by a parallel processing system having a plurality of processing units. More particularly, the described embodiments relate to systems and methods for auto-parallelization of code for execution in the described multi-processor systems. In the technical field of multiprocessor systems, important characteristics and classifications arise inter alia, from way, how processor memory access is handled and whether system processors are of a single type or various ones in the system architecture.

Background of the Invention

In the last years, a fundamental change in computer architecture has been observed that will impact every facet of data processing and operation of every electronic device from cell phone to supercomputer, since it introduces parallelism of unprecedented scale, which confronts traditional technical approaches of code optimization and parallelization. In particular, it creates the technical requirement for computer and processor-specific code parallelization to achieve a true optimization. Whereas the conventional multicore approach (2, 4, and even 32 cores) adopted by the computing industry came slowly to a performance plateau, technology moves to higher performance per watt and per chip area using manycore technology (hundreds or even thousands of cores). However, fully unleashing the potential of the manycore approach to ensure future advances in sustained computational performance will

require fundamental advances in computer architecture and programming techniques which can be even compared to reinventing computing. The new technological trends in the microprocessor industry have important impacts for the design of the next generation of computing systems, in particular of so called High Performance Computing (HPC) systems as the petaflop scale is achieved. The need to switch to a geometric growth path in system concurrency is leading to reconsideration of interconnect design, memory balance, and I/O system design that will have dramatic consequences for the design of future HPC applications and algorithms, in particular the parallelization and architecture specific adaption of future applications and algorithms. The required reengineering of existing application codes will likely be as dramatic and technically challenging as the migration from vector HPC systems to Massively Parallel Processors (MPPs) that occurred in the 90's. Such comprehensive code reengineering took nearly a decade, so there are in the prior art serious concerns about undertaking yet another major transition in the software infrastructure in use.

One of the technical challenges and difficulties comes also from the various different technical fields, suddenly confronted by their interconnected impact on the performance optimization of the new systems, as circuit design, computer architecture, embedded hardware/software, programming languages, compilers, applied math to HPC etc. and the technical object on how current constraints on device physics at the silicon level can be considered to further optimize CPU design, system architecture, and programming models for future and current systems. If these technical challenges cannot be overcome by appropriate systems, in particular code optimization and parallelization systems, this would even lead to the question if multicore (or manycore) or multi-processor is in fact a reasonable response to the underlying constraints to future IC designs. The present invention allows solving the technical need for new optimization systems induced by these changes in the context of computer architecture, system architecture, and programming models for future computing systems, as e.g. HPC systems.

According to the Moore's law, which still holds (likely until 2023 according to latest results), it is possible to integrate twice as many components onto an integrated circuit every 18 months at fixed cost. However, the traditional sources of performance improvements such as instruction level parallelism (ILP) and clock frequency scaling have been flattening since 2003. In particular, the improvements in processor performance as measured by the SPEC benchmark over the period from 1975 to

present has improved by 52 percent per year with remarkable consistency since 1986 (e.g. see J. L. Hennessy, D.A. Patterson, "*Computer Architecture: A Quantitative Approach*", fourth edition, Morgan Kaufmann, San Francisco, 2006). During that period, as process geometries scaled according to Moore's law, the active capacitance of circuits scaled down so that supply voltages could be kept constant or even dropped modestly in order to allow manufacturers to increase clock-speeds. This approach, known as "*constant electric field*" frequency scaling fed the relentless increases in CPU clock-rates over the past decade and a half. However, below the 90nm scale for silicon lithography, this technique began to hit its limits as static power dissipation from leakage current began to surpass dynamic power dissipation from circuit switching. Power density has now become the dominant constraint in the design of new processing elements, and ultimately limits clock-frequency growth for future microprocessors. The direct result of power constraints has been a stall in clock frequency that is reflected in the flattening of the performance growth rates starting in 2002. In the following years, individual processor cores have become nearly a factor of three slower than if progress had continued at the historical rate of the preceding decade. Other approaches for extracting more performance such as Instruction Level Parallelism (ILP) and out-of-order instruction processing have also delivered diminishing returns. Having exhausted other well-understood avenues to extract more performance from a uniprocessor, the mainstream microprocessor industry has responded by halting further improvements in clock frequency and increasing the number of cores on the chip. Estimates show that the number of cores per chip doubled every 18-24 months from there on. Therefore, a new kind of parallelization systems and processor-architecture specific optimizing programming structures are needed to stay ahead of a wave of geometrically increasing system concurrency causing a tsunami of parallelism. The stall in clock frequencies and the industry's comparatively straightforward response of doubling cores has led to the emerge of alternative approaches to computing such as Field Programmable Gate Arrays (FPGAs), Graphics Processing Units (GPU), or dataflow-like tiled array architectures such as TRIPS. The principle impediment to adapting such more radical approach to hardware architecture is that, in the technical field, it is even less known about how to program efficiently such devices for diverse applications than it is known for parallel machines composed of multiple CPU cores.

To run a program in parallel on a multi-core processor or multi-processor system today, its code must be rewritten to add by hand or through tools some OS

parallelizing primitives such as *pthread* (POSIX Threads, known as *pthreads*, provide a parallel execution model structure. It allows a program to control multiple different flows of work that overlap in time. Each flow of work is referred to as a thread, and creation and control over these flows is achieved by making calls to the POSIX Threads API. POSIX Threads is an API defined by the standard *POSIX.1c, Threads extensions IEEE Std 1003.1c-1995)*). Even with high level interfaces like OpenMP or MPI, parallelizing is not an easy job for two reasons: (i) if the resulting code is not enough synchronized, the computation is not deterministic and (ii) if it is too much synchronized, it is not parallel enough. However, threads make programs typically nondeterministic, and rely on programming style to constrain that nondeterminism to achieve deterministic aims. Instead of hand-parallelizing the code, the technical skilled person may rely on a compiler to automatically do, for example, loop vectorization or loop parallelization. It is to be noted that loop-level parallelization is one of the central aspects and technical challenges in HPC applications as loop-sections can introduce large computing demands. Opportunities to exploit parallelism is often in applications where data are stored in random access data structures. However, even simple example can often not be automatically parallelized by prior art compilers. Further, irregular code structures are a second problem. Below, it is detailed discussed how loops are parallelized by the present inventive system, using a just-in-time-loop-parallelization structure.

This leads to a third problem of parallelization which is the memory organization of the data. In the sum example, the array to be summed is declared for example as a global variable. Hence, it is centralized when the computation is distributed. As a result, each thread brings the array pieces it needs from DRAM, where it resides. Cache can help but neighbor cores are slowed down by memory contention and the cache miss rate is impacted by the array distribution. Moreover, in a program updating shared data, keeping caches coherent requires complex hardware which slows down average memory access time. Caches and memory hierarchy, as well as branch predictors, are hardware features that rely on the principle of locality, which in essence is founded on the centralization of data (caches) and fetched code (predictors). When the code and the data are distributed, it is the parallel locality which applies to data. The parallel locality principle is that a consumer should be as close as possible from its producer. The present inventive system also allows to optimize the producer to consumer distance for a parallelized code, which is another way to quantify the parallelization quality.

In the prior art, there are various trials to apply dataflow architecture differently to parallel computing frameworks and to auto-parallelization systems. Dataflow architecture is a dataflow-based computer architecture that directly contrasts the traditional von Neumann architecture or control flow architecture. However, dataflow architectures have no program counter, where in concept, the executability and execution of instructions is solely determined based on the availability of input arguments to the instructions, so that the order of instruction execution is unpredictable, i.e., behavior is nondeterministic. Thus, there is a need to provide dataflow architectures that are deterministic in nature enabling auto-parallelization compilers to manage complex technical tasks such as processor load balancing, synchronization, and accesses to common resources by considering the underlying system architecture.

In general, when processing code is executed in parallel in multi-core or multi-processor systems, this may lead to higher throughput. Multi-processor or multi-core systems (processor and core are for simplicity hereafter simply referred as processor) entail breaking of code into smaller code blocks and efficiently managing the execution of code. In order for the cores or processors to execute in parallel, data to each of the core or processors has to be independent. Instances of a same code block can be executed on several processors simultaneously to improve the through-put. If the processors need data from a previous execution or another process currently performing calculations, then parallel processing efficiency could decrease due to the latency occurring exchange data between and/or of the processor units. In general, when processors enter states in which the execution of a program is suspended or not executed, for whatever reason, and instructions belonging to it are not fetched from memory or executed, those states induce idle states of the processor affecting parallel processing efficiency. Data dependency needs to be considered when scheduling processors. Efficiently managing multiple processors and data dependency for higher throughput is challenging. It is desirable to have a method and system for efficient management of code blocks in computation rich applications. It is to be noted, that the problem of latency also exists in single processor systems, where, for example, latency oriented processor architecture is used to minimize the problem, which are microarchitecture of microprocessor designed to serve a serial computing thread with a low latency. These architectures, in general, aim to execute as many instructions as possible belonging to a single serial thread, in a given window of time, where the time to execute a single instruction completely from fetch to retire stages

may vary from a few cycles to even a few hundred cycles in some cases. However, these techniques do not automatically apply to the latency problem of (massive) parallel computing systems.

Thus, parallel computing systems require efficient parallel coding or programming, where parallel programming becomes a programming paradigm. It comprises on the one hand methods to divide a computer program into individual sections that can be executed concurrently, and on the other hand methods to synchronize concurrent code sections. This is in contrast to classical sequential (or serial) programming and coding. Parallel execution of a program can be supported on the hardware side; the programming languages are then typically adapted to this. For example, parallel programming can be done explicitly by letting the programmer execute program parts in separate processes or threads, or it can be done automatically so that causally independent (parallelizable) sequences of instructions are executed side by side, i.e. parallel. This parallelization can be done automatically by a compiler system if a computer with a multi-core processor or a parallel computer is available as target platform. Some modern CPUs can also recognize such independence (in the machine code or microcode of a program) and distribute the instructions to different parts of the processor in such a way that they are executed simultaneously (out-of-order execution). However, as soon as the individual processes or threads communicate with each other, they are, in that sense, no longer concurrent as a whole since they influence each other, only individual sub-processes being still concurrent with each other. If the order of execution of the communication points of the individual processes or threads cannot be defined accordingly, conflicts can arise, especially so-called deadlocks, when two processes wait for each other (or block each other) or race-conditions, when two processes overwrite each other's results. In the prior art, to solve this problem, synchronization techniques are used, as e.g. mutual exclusion (Mutex) techniques. Though such techniques can prevent race conditions, they do not automatically allow an optimized parallel handling of the processes or threats with a minimal latency of the processor units.

(Micro-)processors are based on integrated circuits, which make it possible to do arithmetic and logical operations based on two binary values (in the simplest 1/0). For this, the binary values must be available for the processor's calculating unit. A processor unit needs to get the two binary values to calculate the result of an expression a = b operand c. The time it takes to retrieve the data for these operations is

known as the latency time. There is a wide hierarchical range for these latency times from the register, L1-cache, memory access, I/O Operation, or a network transfer, as well as processor configuration (e.g. CPU vs. GPU). As every single component has a latency time, the overall latency time for the calculation is a combination mainly of hardware components needed to get the data from one location to the other in a modern computation infrastructure. In modern architectures, also the different software layers (e.g. of an operating system) have large influences. The difference between fastest location for a CPU (or GPU) to get data and the slowest can be huge (in the range of factor $>10^9$). Figure 1 illustrates the formation of latency times in modern computation infrastructures. As figure 1 shows, parallel computing machines have been developed with different distinct architecture. It is important to note, that parallel architecture enhances the conventional concepts of computer architecture with communication architecture. Computer architecture defines critical abstractions (like user-system boundary and hardware-software boundary) and organizational structure, whereas communication architecture defines the basic communication and synchronization operations. It also addresses the organizational structure.

Computer applications are typically written in the top layer, i.e. in a high level language, being based on corresponding programming models. Various parallel programming models are known, as e.g. (i) shared address space, (ii) message passing, or (iii) data parallel programming referring to corresponding multi-processor system architectures. Shared memory multiprocessors are such a class of parallel machines. Shared memory multiprocessor systems give better throughput on multiprogramming workloads and supports parallel programs. In this case, the computer system allows a processor and a set of I/O controller to access a collection of memory modules by some hardware interconnection. The memory capacity is increased by adding memory modules and I/O capacity is increased by adding devices to I/O controller or by adding additional I/O controller. Processing capacity can be increased by implementing faster processors or by adding more processors. As illustrated in figure 2, the resources are organized around a central memory bus. Through the bus access mechanism, any processor can access any physical address in the system. As all the processors are assumed to be or in fact equidistant from all the memory locations, the access time or latency of all the processors is same on a memory location. This is called symmetric multiprocessor system.

The message passing architecture is another class of parallel machines and programming model. It provides communication among processors as explicit I/O operations. The communication is combined at the I/O level, instead of the memory system. In message passing architectures, user communication is executed by using operating system or library calls that perform lower level actions, which includes the actual communication operation. As a result, there is a distance between the programming model and the communication operations at the physical hardware level. Send and receive is the most common user level communication operations in message passing system. Send specifies a local data buffer (which is to be transmitted) and a receiving remote processor. Receive specifies a sending process and a local data buffer in which the transmitted data will be placed. In send operation, an identifier or a tag is attached to the message and the receiving operation specifies the matching rule like a specific tag from a specific processor or any tag from any processor. The combination of a send and a matching receive completes a memory-to-memory copy. Each end specifies its local data address and a pair wise synchronization event. Though, message passing and a shared address space traditionally represent two distinct programming models, each having an own paradigm for sharing, synchronization and communication, the basic machine structures have converged towards a common organization, today.

Finally the data parallel processing is a further class of parallel machines and programming model, also referred as processor arrays, data parallel architecture or single-instruction-multiple-data machines. The main feature of this programming model is that operations can be executed in parallel on each element of a large regular data structure (like array or matrix). Data parallel programming languages are usually enforced by viewing the local address space of a group of processes, one per processor, forming an explicit global space. As all the processors communicate together and there is a global view of all the operations, so either a shared address space or message passing can be used. However, the development of programming models only cannot increase the efficiency of the computer, nor can the development of hardware alone do it. Further, top level programming models necessarily introduce boundary conditions, as e.g. the model specific architecture, given by the programming model requirements. Since a parallel program consists of one or more threads operating on data, the underlying parallel programming model defines what data the threads require, which operations can be performed on the required data, and which order is followed by the operations. Thus, there are limitations on the

optimization of machine codes for multi-processor systems, due to the boundaries of the underlying programming model. A parallel program must necessarily coordinate the activity of its threads to ensure that the dependencies between the programs are enforced.

As figure 1 shows, parallel computing machines have been developed with different distinct architecture, each giving raise to different formation of latency times in their computation infrastructures. One of the most common multi-processor systems is the shared memory multiprocessors systems. Essentially, there are three basic architectures known for shared memory multiprocessors systems: (i) Uniform Memory Access (UMA), (ii) Non-uniform Memory Access (NUMA), and (iii) Cache Only Memory Architecture (COMA). In the UMA-architecture (see figure 3), all the processors share the physical memory uniformly. All the processors have equal access time to all the memory words. Each processor may have a private cache memory. Same rule is followed for peripheral devices. When all the processors have equal access to all the peripheral devices, the system is called a symmetric multiprocessor. When only one or a few processors can access the peripheral devices, the system is called an asymmetric multiprocessor. In the NUMA multiprocessor architecture (see figure 4), the access time varies with the location of the memory word. The shared memory is physically distributed among all the processors, called local memories. The collection of all local memories forms a global address space which can be accessed by all the processors. Finally, the COMA multiprocessor architecture (see figure 5) is a special case of the NUMA multiprocessor architecture. In the COMA multiprocessor architecture, all the distributed main memories are converted to cache memories. The COMA architecture can also be applied to distributed memory multi-computers. A distributed memory multicomputer system consists of multiple computers, typically denoted as nodes, inter-connected by message passing network. Each node acts as an autonomous computer having a processor, a local memory, and sometimes I/O devices. In this case, all local memories are private and are accessible only to the local processors, thus such machines are also called no-remote-memory-access (NORMA) machines. Other known multi-processor architectures are e.g. multivector computers and Single Instruction, Multiple Data (SIMD) parallel computers, Parallel Random-Access Machines (PRAM) and parallel computers based on Very Large-Scale Integration (VLSI) chip etc., all having a different multiprocessor architecture and infrastructure characteristic. In summary, since different multiprocessor architectures give raise to different formation of latency times in their computation infrastructures, the development of programming

models only cannot increase the efficiency of the computer, nor can the development of hardware alone do it.

As mentioned above, code parallelization can also be performed automatically by a compiler system, if a computer with a multi-core processor or a parallel computer is available as target platform. Such automatic parallelization, also called auto parallelization, denotes the converting of sequential code into multi-threaded and/or vectorized code in order to use multiple processors simultaneously e.g. in a shared-memory multiprocessor (SMP) machine. With prior art systems, fully automatic parallelization of sequential programs is technically challenging because it requires complex program analysis and because the best approach can depend upon parameter values that are not known at compilation time. The programming control structures on which auto parallelization by compiler systems places the most focus are loops, because, typically, most of the execution time of a program takes place inside some form of loop. There are two main approaches to parallelization of loops: pipelined multi-threading and cyclic multi-threading. The compiler structure for automatic parallelization normally comprises a parser, an analyzer, a scheduler, and a code generator. The parser of a compiler system covers the first processing stage where e.g. a scanner reads the input source files to identify all static and extern usages. Each line in the file will be checked against pre-defined patterns to segregate into tokens. These tokens will be stored in a file which will be used later by the grammar engine. The grammar engine will check patterns of tokens that match with pre-defined rules to identify variables, loops, control statements, functions etc. in the code. In a second stage, the analyzer identifies sections of code that can be executed concurrently. The analyzer uses the static data information provided by the scanner-parser. The analyzer first detects all the totally independent functions and mark them as individual tasks. The analyzer then finds which tasks have dependencies. In the third stage, the scheduler will list all the tasks and their dependencies on each other in terms of execution and start times. The scheduler will produce the optimal schedule in terms of number of processors to be used or the total execution time for the application. In the fourth and final stage, the scheduler generates a list of all the tasks and the details of the cores on which they will execute along with the time that they will execute for. The code generator then inserts special constructs in the code that will be read during execution by the scheduler. These constructs will instruct the scheduler on which core a particular task will execute along with the start and end times.

If a cyclic multi-threading parallelizing compiler is used, the compiler tries to split up each loop so that each iteration of the loop can be executed on a separate processor concurrently. During auto parallelization, the compiler typically conducts two passes of automated assessment before actual parallelization in order to determine the following two fundamental preconditions for parallelization: (i) In a first pass, based on dependence analysis and alias analysis, is it safe to parallelize the loop?, and (ii) in a second pass, based on estimation (modeling) of the program workload and the capacity of the parallel system, is it worthwhile to parallelize it? The first pass of the compiler performs a data dependence analysis of the loop to determine whether each iteration of the loop can be executed independently of the others. Data dependence can sometimes be dealt with, but it may incur additional overhead in the form of message passing, synchronization of shared memory, or some other method of processor communication. The second pass attempts to justify the parallelization effort by comparing the theoretical execution time of the code after parallelization to the code's sequential execution time. It is important to understand, that code does not always benefit from parallel execution. The extra overhead that can be associated with using multiple processors can eat into the potential speedup of parallelized code.

If a pipelined multi-threading parallelizing compiler is used for auto parallelization, the compiler tries to break up the sequence of operations inside a loop into a series of code blocks, such that each code block can be executed on separate processors concurrently.

There are many parallel problems that have such relatively independent code blocks, in particular systems using pipes and filters. For example, when producing live broadcasting, many different tasks must be performed many times a second.

A pipelined multi-threading parallelizing compiler tries to assign each of these operations to a different processor, normally arranged in a systolic array, inserting the appropriate code to forward the output of one processor to the next processor. For example, in modern computer systems, one of the focuses lies on using the power of GPU's and multicore systems to compute such independent code blocks (or independent iterations of a loop) at runtime. The memory accessed (whether direct or indirect) can be then marked for different iterations of a loop and can be compared for dependency detection. Using this information, the iterations are grouped into levels

such that iterations belonging to the same level are independent of each other, and can be executed in parallel.

In the prior art, many compiler for automatic parallelization exist. However, most modern prior art compilers for automatic parallelization depend on the use of Fortran as high level language, i.e. are applicable to Fortran programs only, because Fortran makes stronger guarantees about aliasing than languages such as C. Typical examples of such prior art compilers are (i) Paradigm compiler, (ii) Polaris compiler, (iii) Rice Fortran D compiler, (iv) SUIF compiler, And (v) Vienna Fortran compiler. Further disadvantages of the automatic parallelization by the prior art compilers lies in the fact, that a high optimization of the code is often difficult to achieve due to the fact, that (a) dependence analysis is hard for code that uses indirect addressing, pointers, recursion, or indirect function calls because it is difficult to detect such dependencies at compile time; (b) loops have often an unknown number of iterations; (c) accesses to global resources are difficult to coordinate in terms of memory allocation, I/O, and shared variables; and (d) irregular algorithms that use input-dependent indirection interfere with compile-time analysis and optimization.

One important task of a compiler is to try to handle the latency times efficiently. Compiling is the transcription from human readable and so-called higher languages (such as C, python, java, etc.) to assembler / processor code, which then only consists of the available instructions on a given processor. As already discussed, modern applications with a large demand for data or computation have to be targeted for appropriate infrastructure and a lot of different latency times are introduced - and currently only parts of it can be solved by prior art compiler optimization techniques.

For every level of complexity (hardware components), the solutions have historically been developed and evolved, from compiler optimization techniques to multithread libraries for concurrent data structures to prevent race-conditions to vectorization of code, to GPU systems with corresponding programming languages (e.g. OpenCL (Open Computing Language) to frameworks such as "TensorFlow" to distribute computations by programmers, to big data algorithm such as the "MapReduce", where MapReduce is a programming technique and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. In the field of high-performance computing, theoretical,

mathematics-based techniques to split large matrices to special gridding techniques for finite difference or element methods were developed and defined. This includes protocols, e.g. in cluster infrastructures Message Passing Interface (MPI) supports the transfer of data to different processes over an infrastructure.

As discussed above, the list of prior art optimization techniques is long. But from a system theory perspective, the question is more or less always the same: How can the code (any code) interact most efficiently with the latency times in a complex hardware infrastructure? Compilers work well when using a single CPU. As soon as the hardware complexity increase a compiler cannot really render the code parallel. The parallelization becomes merely CPU approximated, e.g. by introducing microprocesses. Hardware industry for CPUs, GPUs and clusters thereof mainly focuses on their specific domains, developer and research focus on implementation techniques and framework development and have so far not moved into the field of a more generic (cross- industry) approach. Further, the authors Ruggiero, Martino; Guerri, Alessio; Bertozzi, Davide; Milano, Michaela; Benini, Luca, revealed in their pamphlet: A Fast and Accurate Technique for Mapping Parallel Applications on Stream-Oriented MPSoC Platforms with Communication Awareness, International Journal of Parallel Programming, Vol. 36, No. 1, Feb 08, the partitioning of algorithms that process data streams onto different processor cores. The authors model is a simple communication network with a simple, additive communication model between processor cores. This does not allow realistic conclusions about the actual communication load caused by partitioning on multiple cores.

Generally, known processor manufacturer are focused on their processors and related hardware components, whereas other developers, for example, the research group for High Performance Computing (HPC) are focused on the numerical methods and the use of libraries. Currently, there is no attempt to solve the question regarding an optimization of a compiler system from a system theoretical perspective by accessing the latency dynamics resulting from a given source code. The source code in the prior art merely consists of a series of statements which result in read and write direct for a given target infrastructure.

The prior art document M. Kandemir et al. "*Slicing Based Code Parallelization for Minimizing Inter-processor Communication*", 2009 International conference on compilers, architecture, and synthesis for embedded systems (cases

'09), Grenoble, France October 11-16, 2009, p. 87-96, discloses a system for auto-parallelization that aims in minimizing inter-processor communication in distributed memory multicore architectures by applying the concept of iteration space slicing, i.e. this prior art system is based on an iterative approach. The disclosed system does this by the portioning of output array by iteratively determining the partition of other arrays in the application code, i.e. by iterative determination of array parts, wherein the information is taken iteratively from the previous sliced array parts. In code parallelization, slicing denotes the process of extracting statements form a program that potentially have an influence on a specific statement of interest, which is the slicing criterion (see e.g. J. Krinke, "*Advanced slicing of sequential and concurrent programs*,"*20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, 2004, pp. 464-468). These slicing techniques showing similar effects as the prior art systems relying on points data/control dependences (see e.g. J.L. Hennessy, D.A. Patterson "Computer Architecture", Fifth Edition: A Quantitative Approach, The Morgan Kaufmann Series in Computer Architecture and Design, fifth edition, p. 150ff) and data flow analysis (see e.g. Gary A. Kildall "A unified approach to global program optimization", proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL '73). Association for Computing Machinery, New York, 1973, USA, 194–206). Using iteration space slicing, these systems are able to assess which iterations of which statements is affecting the values of a given set of elements from a specific array A. Thus, iteratively, the system returns the set of loop operations to be assigned to a processor p e.g. from a loop nest s by relying on a specific set of data elements accessed by processor p from array A. Further, the prior art document Fonseca A. et al, "*Automatic Parallelization: Execution Sequential Programs on a Task-Based Parallel Runtime*", International Journal of Parallel Programming, April 2016 discloses another system for automatically parallelizing of sequential code for use in multicore architectures. The system discloses to use data-groups and memory layouts and then check the dependencies relying on task parallelism. Thus, in order to automatically parallelize the program, it is necessary for the system to analyze the memory accessed to assess possible dependencies between parts of the program. For instance, in an example of auto-parallelization of a code for Fibonacci sequencing, the disclosed system assesses the cost of creating a new task as higher than the cost of executing the method for a low input number. This assessment will then be used as main requirements for the position of the task during the auto-generation of the parallelized code, where the assessment is conducted by a specific function relying on a set of seven requirements to find the best position. Finally, this

function outputs so called hard dependencies, which are the instructions after which a task can be introduced, and so called soft dependencies giving a set of already defined tasks which the current task will have to wait to execute. The parallelization is completed when all tasks are instantiated having a specified position, wherein a task is marked for execution by awaiting the execution of the current tasks and reading their results. Finally, US 2008/0263530 A1 discloses a system for converting application code into optimized application code or into execution code suitable for execution on a computation architecture comprising at least a first and a second level of data memory units. When scheduling instructions, a principle of locality, also called locality of reference, is used. This relates to the phenomenon that a same value or related storage location is frequently accessed. Different types of locality of reference are to be distinguished. In temporal locality, a resource that is referenced at one point in time is referenced again soon after-wards. In spatial locality, the likelihood of referencing a storage location is greater if a storage location near it has been recently referenced. Programs and systems which exhibit locality exhibit predictable behavior, and thus provide opportunities for code designers to improve performance through prefetching, precomputing and caching of code and data for future use. For this kind of data assessment optimization, the disclosed system accesses locality before layout locality, whereby for data, which is accessed again and again, accesses are brought together in time where possible when data transfer operations happen, and data which is accessed one after the other is brought together in space where possible. Therefore, in a first process (access locality) a partial fix is done, giving a range of options. In a second process (layout locality) one option is picked from the pre-defined range. This picking of one option may be done based on a cost function. As an embodiment variant, the system also addresses the problem of parallel data transfer and storage exploration by focusing on those parts of the application code with data parallel loops. A conversion structure allows to address both data-level aspects of different levels of memory units, such as back-ground memories, fore-ground memories, registers, and on functional units.

In summary, although parallelizing compilers exist, they typically focus on independent levels of parallelism such as instruction-level parallelism (ILP) targeting VLIW (Very Long Instruction Word) and EPIC (Explicitly Parallel Instruction Computing) architectures, data, and vector parallelism such as high-performance Fortran and MMX/SSE-capable compilers, or thread-level parallelism (TLP) as in the case of compilers supporting the OpenMP API or the PGI Accelerator Compilers for GPUs. Even

more sophisticated, architecture-centric approaches such as IBM's Octopiler, still the need for manual performance tuning exists; therefore, all software-development processes have to be manually adapted to the new prerequisites. All kinds of multilevel parallelism, finer granularities, and nested hierarchical memory systems combined with software-managed or user-managed data transfers need to be utilized and expressed by the algorithms. In addition, applications need to be designed with respect to scalability across several processor generations and concepts. For all steps of application mapping (e.g., in numerical simulation), comprehensive hardware knowledge is required to achieve optimal throughput. An intricate trade-off exists between performance, productivity, portability, and flexibility of an implementation. In the present inventive system and method, hardware-aware computing is approached by providing the best combination of schemes, parallel algorithms, and platform-specific implementations for a given source code of an application.

Further, current, prior art auto-parallelization systems and programming approaches towards multicore systems – and especially heterogeneous parallel systems – unfortunately do not suit the specific needs of such systems: high-level parallel programming models deliver an abstract view as required by today's complex applications but do not feature fine-grained control over hardware mapping, resulting in poor hardware resource usage. Strict hardware-aware approaches, in term, enable such fine-grained control but will put the focus to hardware mapping. The inventive system and method being based on a hybrid approach provides a new technical combination of both aspects.

As further discussed below, conventional, prior art, parallel programming approaches can be divide into the following three fields that are shared-memory, message-passing, and data-parallel approaches with according standardized programming environments such as (in order of fields mentioned) OpenMP (Open Multi-Processing) for targeting compiler-exploitable TPL (Task Parallel Library), MPI (Message Passing Interface) for addressing manually exploited thread-level and pipeline parallelism, and high-performance Fortran for compiler-supported exploitation of Data Parallelity (DP). These programming models support only marginal flexibility and typically do not enable fine-grained architecture mapping. Of these three, message passing provides the highest level of controlling the architecture mapping of an application. On the downside, it forces the programmer to detailed partitioning and orchestration. One of the larges disadvantages is further, that hardware heterogeneity

is typically not expressed by any of these models as this is outside their focus. Certain flexibility is supported by OpenMP allowing dynamic change of the number of threads to be created. Partitioned Global Address Space (PGAS) is a programming interface that defines a global address space on a possibly distributed system focusing on reference-locality exploitation: with PGAS, portions of the shared-memory space may have an affinity for a particular thread. Examples of this model are Unified Parallel C and Co-array Fortran but also recent and industry-driven approaches such as Chapel (Cray) and X10 (IBM).

Especially heterogeneous platforms require fine-grained ways of matching application code to the given platform on the lowest hardware level: one prior art example is the Cell BE forcing the programmer to explicitly partition the program into individual chunks to be executed on the single vector processing units, the SPEs. Communication must be explicitly formulated using so-called direct memory access streams, enabling the individual SPEs to fetch required data and write back result data. This careful matching of computation and communication may lead to speedups of orders of magnitude compared with a naïve implementation. It is however a quite tedious, error-prone, and time-consuming task. Hardware-aware programming techniques are also required on conventional homogeneous multicore platforms to minimize communication overhead, for example, by data-locality optimizing and appropriate prefetching. With current and upcoming multicore architectures, this becomes even more prevalent as such architectures feature distinct schemes of cache sharing and interconnection technologies.

Another problem of all discussed prior art approaches is the necessity of additional run-time layers and often language-centric extensions. They furthermore do not regard an application's required such as real-time requirements or computation accuracy. A different prior art approach is based on carefully extending existing system layers. However, it is inherently language agnostic and operating system agnostic and not compatible to existing parallel programming models. Further, overhead does solely appear for triggered options like performance measurements and function profiling for guided execution and the profiling overhead still is about an order of magnitude lower than often required.

To achieve real, hardware-optimized parallelization, all parts of the application code need to be compliant with multilevel parallel units, a hierarchical and

potentially nested memory subsystem, and heterogeneous components of the computing units. All aspects need to be expressed explicitly in the implementations: in the prior art, there is only little software support and few mechanisms available that may assist automatic and optimized utilization of resources, as well as hide away hardware details without affecting performance. Although heterogeneity in hardware is growing, it still lacks expression within the used algorithms and applications. Current programming techniques mainly rely on minimally invasive approaches where local parts of the application are identified for acceleration and offloaded to particular computing engines. For the overall benefit, additional communication via narrow bottlenecks has to be taken into account. However, prior art solutions are often isolated and not portable. Hardware-specific optimization techniques include arrangement of computations (such as vectorization, loop unrolling, re-ordering, branch elimination), optimizing of data structures (array-access patterns, memory alignment, data coalescing), and optimization of data transfers (blocking for spatial and temporal locality, caching and cache-bypassing, in-place algorithms). This should include the parameter space since the parameter space becomes increasingly difficult to manage. With the known prior art systems, there is no way around auto-tuners searching for (Pareto-)optimal settings of implementation parameters. For selected kernels, autotuning may provide results, however, typically on the price of lack of portability across platforms.

Summary of the Invention

It is one object of the present invention to provide compiler system for multi-processor systems and multi-computer systems compiling program codes to machine codes with optimized latency of the processing units of the multi-processor systems, thereby efficiently managing multiple processors and data dependency for higher throughput, and not having the disadvantages of the prior art systems, as discussed above. In particular, it is an object of the present invention to provide a system and technique that can be used to achieve the highest performance in multi-processor machines through auto parallelization, thereby optimizing the exploitation of low-level parallelism (temporal and spatial) at the level of the processing of machine instructions. It is a further object of the invention to overcome disadvantages of prior art techniques overcoming their limitations of dealing with parallelization sections which are typically

restricted to specific systems in mind like loop or particular section of code. The auto-parallelization system should be able to optimize identifying opportunities for parallelization as a critical step while generating multithreaded application.

According to the present invention, these objects are achieved, particularly, with the features of the independent claims. In addition, further advantageous embodiments can be derived from the dependent claims and the related descriptions.

According to the present invention, the above-mentioned objects for a symmetric auto-compiler system for hardware-optimized auto-parallelization of a program code for execution by a multi-core or multi-processor parallel processing system having a plurality of processing units simultaneously processing instructions on data in the parallel processing system by executing the program code, and corresponding method are achieved, particularly, in that the auto-compiler system comprising means for transforming a serial source code of the program code written in a programming language into a parallel processing machine code comprising a number of instructions executable by the plurality of processing units of the parallel processing system or controlling the operation of the plurality of processing units, wherein the parallel processing system comprise memory units comprising at least main execution memory units comprising a plurality of memory banks for holding data of at least parts of the processing code, and transition buffer units comprising high speed memory for storing starting locations of the processing code and data segments at least comprising branch or jump instructions and/or used memory references and data values, wherein the main execution memory units provide a slower access time than the transition buffer units, wherein the execution of the processing code by the parallel processing system comprises the occurrence of latency times, the latency times being given by idle time of a processing unit between transmitting data back after having processed a specific block of instructions of the processing code on the data by the processing unit and receiving data necessary for execution of a consecutive block of instructions of the processing code by said processing units, in that the compiler system comprises a parser module for translating the serial source code into a code with a flow of elementary instructions executable by the processing units, the elementary instructions selectable out of a limited, processing-unit-specific set of elementary instructions and the elementary instructions comprising exclusively elementary arithmetic and logic operations and/or elementary control and memory operations for

the number of processing units, in that the parser module comprises means for partitioning the code of said elementary instructions into computation block nodes, each computation block node consisting of a smallest possible segmentation of a non-further decomposable sequence of elementary instructions of the code processable by a single processing unit, the smallest possible segmentation of the elementary instructions being characterized by a sequence of elementary instructions framed by consecutive read and write instructions, said sequence being not further decomposable by smaller sequences of elementary instructions between consecutive read and write instructions, and the read and write instructions needed to receive data required for processing said sequence of elementary instructions by the processing unit and transmit back data after processing by the sequence, in that the compiler system comprises a matrix builder for generating matrices out of computation chains portioned from the code, the matrices comprising computation matrices and transfer matrices and task matrices, wherein each column within a computation matrix comprises computation block nodes being simultaneously processable based on executability of read and write instructions transferring data required for processing of the computation block nodes, wherein the transfer matrix contains to each computation block nodes transfer and process properties at least characterizing data transfer properties from one to a consecutive computation block node at least comprising data size of transmitted data and identification of a source computation block node and target computation block node of the data transfer and/or process characteristics on one of the plurality of processing units, in that tasks are formed by the matrix builder wherein in case of computation block nodes each having a different read associated, the tasks of a task matrix are formed by splitting the computation block nodes of a column of the computation matrix evenly to the number of the plurality of symmetric processing units forming for each of the plurality of processing unit per column of the computation matrix one task, and splitting remaining computation block nodes based on a predefined scheme to at least a part of said tasks, and wherein in case of computation block nodes at least partially having a read with identical data transferred, the tasks are formed by minimizing the number of reads uniformly or essentially uniformly over the number of processing units and/or if a predefined offset value is exceeded by minimizing an integrated processing time uniformly over each of the processing units, and in that the compiler system comprises a code generator for generating the parallel processing machine code for the plurality of processing units with an optimized aggregated latency time based on the computation chains given by the optimized task matrix. Additionally the compiler

system can e.g. comprise an optimizer module using a matrix optimization technique minimizing an aggregated occurring latency time integrating all occurring latency times by providing an optimized structure of the tasks within the task matrix, wherein each column of the task matrix forms a computation chain by one or more tasks creating an ordered flow of computation block nodes to be executed by one of the plurality of processing units. If matrices or tensors are used, the optimization by means of the optimizer module can e.g. be based on numerical matrix optimization technique (or more general numerical tensor optimization technique, respectively). Technically, the present optimization problem can be formulated by using tensors and/or matrices and obtain in this way matrix/tensor field optimization problems. For linear optimization, matrices and linear programming can e.g. be used by the optimizer module. For certain applications of the present invention, the concept of tensor can e.g. be technically instrumental. In optimization tensor technics are able to solve systems of nonlinear relations and equations and for unconstrained optimization using second derivatives. Tensor methods can be used as general purpose-methods intended especially for problems where the Jacobian matrix at the solution is singular or ill-conditioned. The tensor methods can also be used for linear optimization problems. An important feature of tensors is that their values do not change when they cause regular nonlinear coordinate transformations, and thus, this notion can technically be useful for the characterization of structural properties not depending on regular nonlinear coordinate transformations. Thus, the tensor optimization can also be applied within the framework of nonlinear optimization. However, it has to be noted, that one of the technical advantages of the present invention is, that all matrices, know up-to-now, are, in contrast to the prior art optimization techniques in the field of auto-parallelization of source code, linear optimization approaches, while the prior art systems mainly must rely on non-linear optimization. Optimization, herein, denotes the problem of finding a set of inputs to an objective function that results in a maximum or minimum function evaluation. For this technically challenging problem, for example, also various machine learning algorithms can be used with the optimizer module from fitting logistic regression models to training artificial neural networks. If the optimizer module is realized by realized machine learning structures, it can be formulated it usually can be provided by using continuous function optimization, where the input arguments to the function are real-valued numeric values, e.g. floating point values. The output from the function is also a real-valued evaluation of the input values. However, as an embodiment variant, also optimization functions can be used taking discrete variables, i.e. providing a combinatorial optimization problem. To technically select the best optimization

structure, for example, one approach can be to group the selectable optimization structures based on the amount of information available about the target function that is being optimized that, in turn, can be used and harnessed by the optimization algorithm. It is clear, that the more information is available about the target function, the easier the function is to optimize by machine learning, for sure, dependent from the fact whether the available information can effectively be used in the optimization. Thus, one selection criterion can e.g. be related to the differentiable target function by the question, if the first derivative (gradient or slope) of the function can be calculated for a given candidate solution or not. This criterion partitions the available machine learning structures into those that can make use of the calculated gradient information and those that do not, i.e. machine learning structures that use derivative information, and machine learning structures that do not use derivative information. For the application, where differential object functions can be used, it has to be noted, that herein, differentiable function denotes a function where the derivative can be generated for any given point in the input space. The derivative of a function for a value is the rate or amount of change in the function at that point, which is also referred as the slope. The first-order derivative is defined as the slope or rate of change of an objective function at a given point, where the derivative of the function with more than one input variable (e.g. multivariate inputs) is referred to as the gradient. Thus, the gradient can be defined as derivative of a multivariate continuous objective function. A derivative for a multivariate objective function is a vector, and each element in the vector can be called a partial derivative, or the rate of change for a given variable at the point assuming all other variables are held constant. Further, the partial derivative can be defined as element of a derivative of a multivariate objective function. Then, the derivative of the derivative of the objective function can be generated, that is the rate of change of the rate of change in the objective function. This is referred to as the second derivative. Thus, the second-order derivative can be defined as rate at which the derivative of the objective function changes. For the present case of a function that takes multiple input variables, this is a matrix and is referred to as the Hessian matrix, where the Hessian matrix is defined as second derivative of a function with two or more input variables. Simple differentiable functions can be optimized analytically using known calculus. However, the objective functions may not be solvable analytically. The used optimization can be significantly easier if the gradient of the objective function can be generated. Some machine learning structures that are able to use gradient information and may be used for the present

application include: Bracketing algorithms, local descent algorithms first-order algorithms, and second-order algorithms.

The invention has, inter alia the advantage to provide and achieve a massive optimization based in the lowest possible code structure, reducing high programing language code to a few elementary instructions, which are, in respect to their data input and output points, not further reducible due to the limited set of machine instructions running on a CPU/microprocessor. The elementary instructions comprise, for example, e.g. (i) arithmetic operations: +, -, *, / -> in applied numerical applications, i.e. mathematical operations, such as integrating or differential analysis are reduced to these elementary instructions, (ii) logical operations: AND, OR, etc., (iii) variable and array declarations, (iv) compare operations: same, larger, smaller, etc., (v) code flow: jump, call, etc., (vi) if (condition) {codeA} else {codeB}, and (vii) loop (condition). The interactions between today's modern higher languages (e.g. python, C, java, etc.) and the limited resources of processor instructions can be analyzed and made accessible by creating a 'mapping' of read and write of 'data points' by their operations. In other words, by mapping the interactions of the read and write of the single instructions using an appropriate representation, which can, in addition, also be represented graphical, they can be made available for numerical optimization techniques, which leads to the ability to parallelize source code automatically resulting always with a runnable concurrent code. There are several approaches to access these interactions, but none that consequently maps the read and write pattern of a source code to the data introduced by the programmer's choice of variable definition and then goes on to extract the required sequential chains and introducing potential communication patterns, which makes "mapping" the code to a wide range of hardware infrastructure. This method discloses a new approach of "fitting" source code to a given hardware infrastructure over all levels (FPGA, CPU, GPU, clusters, etc.).

The invention has further the advantage the disclosed method and system can solve known technical problems, such as resolving nested loops with arrays solving PDEs (Partial Differential Equations) with a new perspective or well-known problems occurring in optimizing steps of SOTA compilers. The method gives a new perspective on codes and this new scope is based on physical effects occurring in all classical computation infrastructures, resulting in a generic way to map computations to a given hardware structure and derive a concurrent representation of a code on a given hardware or an ideal hardware for a given code. This is based on the consequence

keeping all dependencies of "reading" and "writing" of the introduced data nodes and building chains of instruction depending on these dependencies. By extracting these chains of instructions, a minimal bit-size to compute the instruction chains for a target platform are extracted. As every chain of instructions can be expressed as combinational circuit (output depends only on the present inputs), these series of instructions represent combinational circuit blocks. Chain of instructions in the same column can be processed parallel only limited by the delays given by the physical properties, respectively limits of binary computing (delay between applying voltage to stable result).

The resulting computation block nodes and their flow graph gives back a well-formed base in matrices, which result in a generic way of getting back code applicable to different computational units (e.g. CPUs, GPUs, FPGAs, micro-controller, etc.). The method has a new perspective on ICT software and hardware interaction following the principles of system theory. This results in a method, which can bring novelty solutions in a wide range of fields, as:

(a) Adaptive hardware – FPGAs (Field Programmable Gate Array)/ ACIDs (Atomicity, Consistency, Isolation, Durability): The invention decomposes code into chains with instructions, which obviously represent logic elements in Integral Circuits. As the method gives a generic form to optimize combination of computing and communicating, it can be used to optimize the group of instructions based on the same «bit-pattern» / «signal» and lead to new approaches e.g. to optimize the transfer, respectively automatically of software to FPGAs or close the gap from code to chip floor-planning.

(b) ML (Machine Learning)/AI (Artificial Intelligence): Machine Learning and Artificial Intelligence codes need a lot of resources, especially in training phases. The method can be used for example to (i) optimize already known codes, (ii) support code development, which adapt their complexity during runtime and therefore are hard to parallelize in advance (as the method always result in an optimized code), (iii) support upcoming, not on Neuronal Network based approaches (e.g. Genetic Algorithms, see e.g. *Inside HPC Special Report, AI-HPC is Happening Now* by R. Farber)

(c) HPC (High Performance Computing) applications: As the method can translate code from e.g. python to a C code with MPI support libraries it can close gaps existing e.g. in different research fields (HPC to AI development). Another application could be adaptive mesh refinement implementations, used in numerical model software packages for engineering applications, weather prediction models, etc.. Or it can be used to combine models with different spatial and temporal resolutions (e.g. Computational Fluid Dynamic models and Agent Based models, etc.) and improve existing software packages in different fields, such as modelling and analytic software packages.

(d) Automated business processes: The invention can also be used in process management, the decision if a unit should work on a task or transmit it to another unit is a well-known problem. For this problem the method shows an approach.

(e) Cloud, desktop operation system, virtual machines, deployment in general: Having a generic approach available, which «reduces» a code on the elementary needed operation and possible concurrent options, supports a wide range of solutions in the interface between software and hardware. This interface occurs obviously especially for any form of operating systems, virtualization and / or deployment of software or more concrete e.g. for virtualization solutions for cloud infrastructures, operating systems (with multicore systems), virtual machines to support mixing of different operating systems, or similar examples.

(f) Heterogeneous platforms, (i)IoT and edge computing: Heterogeneous platforms dominating the given situation in different fields, such as IoT projects, autonomous driving, combined mobile and cloud applications and other forms of application running with and / or on a mixed hardware infrastructure. The method can adapt a code to decide how to optimal distribute data, computations and / or data communication on a platform. Furthermore, she can incorporate in the process of deploying / developing software for different properties of the hardware components of a given network of computing units and

optimize a code to fulfil target properties, such as e.g. reducing latency for some parts of a software system.

(g) Embedded systems: Embedded systems have high requirements for e.g. power consumption or other specific adaption for a given code, such as only reduced instruction sets on some micro-processors or similar challenges. The method can support this mapping directly, as she can optimize for given physical properties and results therefore in a most efficient code representation for any given code.

(h) Self-optimizing algorithms: The invention allows a complete autonomous circle, meaning an algorithm can optimize itself on a given platform without any by-hand interaction. This enables new applications and fields, not known up till now.

Brief Description of the Drawing

The present invention will be explained in more detail, by way of example, with reference to the drawings in which:

Figure 1 shows a diagram schematically illustrating the formation of latency times under different modern computation infrastructures. (Micro-) processors are based on integrated circuits allowing to perform arithmetic and logical operations based on two binary values. For this, the binary values must be available for the processor's calculating unit. A processor unit needs to get the two binary values to calculate the result of an expression a = b operand c. The time it takes to retrieve the data for these operations is known as the latency time. There is a wide hierarchical range for these latency times from the register, L1-cache, memory access, I/O Operation, or a network transfer, as well as processor configuration (e.g. CPU vs. GPU). As every single component has a latency time, the overall latency time for the calculation is a combination mainly of hardware components needed to get the data from one location to the other in a modern computation infrastructure.

Figure 2 shows a block diagram schematically illustrating shared memory multiprocessors as one class of parallel machines giving basis to the shared address programming model as top layer parallel programming. In the shared memory multiprocessors architecture, the assumption is taken that in this computer system a processor and a set of I/O controller are allowed to access a collection of memory modules by some hardware interconnection. The memory capacity is increased by adding memory modules and I/O capacity is increased by adding devices to I/O controller or by adding additional I/O controller. Processing capacity can be increased by waiting for a faster processor to be available or by adding more processors. All the resources are organized around a central memory bus. Through the bus access mechanism, any processor can access any physical address in the system. As all the processors are assumed to be or in fact equidistant from all the memory locations, the access time or latency of all the processors is same on a memory location. This is called a symmetric multiprocessor.

Figure 3 shows a block diagram schematically illustrating an UMA-architecture, In the UMA-architecture, all the processors share the physical memory uniformly. All the processors have equal access time to all the memory words. Each processor may have a private cache memory. Same rule is followed for peripheral devices. When all the processors have equal access to all the peripheral devices, the system is called a symmetric multiprocessor. When only one or a few processors can access the peripheral devices, the system is called an asymmetric multiprocessor.

Figure 4 shows a block diagram schematically illustrating a NUMA multiprocessor architecture. In the NUMA multiprocessor architecture, the access time varies with the location of the memory word. The shared memory is physically distributed among all the processors, called local memories. The collection of all local memories forms a global address space which can be accessed by all the processors.

Figure 5 shows a block diagram schematically illustrating a COMA multiprocessor architecture. The COMA multiprocessor architecture is a special case of the NUMA multiprocessor architecture, where all the distributed main memories are converted to cache memories.

Figure 6 shows a block diagram schematically illustrating the different scopes of the units "basic blocks" (or "block units" respectively) and "computation block

nodes" as schematic examples. "Computation block nodes" as used in the present application differ essentially from those block units, used in SOTA (State-Of-The-Art) compilers.

Figure 7 shows a block diagram schematically illustrating a simplified, but more realistic example, where the inventive method exploits the fact, that a logarithmic instruction (as an example on a modern CPU: FYL2X) needs much longer than a floating point addition and/or multiplication instruction (e.g. FADD, FMUL). The inventive method adds the statement 'x:=a+b' und 'y:=a*b' to two different computation block nodes (cbn), as both are based on the same information 'a', and 'b'. The 'log2(x)' statement is appended to the 'x = a+b' as it uses the information 'x'. The information 'y' is getting transferred, so the information is available to both, independent computation-chains.

Figure 8 shows an example how the code from Fig. 7 is split to machine code executable on two computation units synchronized by any form of IPC (it has to be annotated, that IPC is meant more to be a communication between two computing units than "InterProcessCommunication"). The matrix indicates the needed clock cycles needed to compute the different machine instructions as an example how to get a value for the time to compute the instructions. In this form, it is visible in figure 8, that unit 2 can compute the long running instruction of the statement 'log2(x)' and unit 1 processes the loop with incrementing a ('a = a+1'), see figure 7.

Figure 9 shows a block diagram schematically illustrating the inventive method forming out of a flow-graph of computation block nodes two matrices, the herein called "numerical matrices" (it is to be noted, that as a variant "numerical" as may also refers to text data for particular cases, since all data on an IT infrastructure can be understood as numerical on a low level, e.g. binary). One contains the instruction chains and in the other the possible transfer properties (from and to other computation block nodes). Therefore, the code extracted from the matrices always forms a pattern "compute -> communicate", as can be seen in detail in the following passages. Obviously, if code is mapped to one unit, the communicate part would vanish (and the computations (=chains of instructions) would sum up) and the inventive method would be reduced to the approach with "basic blocks", respectively can be handled with a SOTA compiler Furthermore, the matrices which represent a much more scalable form of accessing information than graphs, respective show the well-formed nature of the control flow graph. The definition of the computation block nodes (cbns)

also indicates the generic nature of the inventive matrices: In one matrix, each row has an independent flow of instructions (= computations) and in the other, the needed transfers (= communications) to other cbns. In consequence, this ensures that a) no further information is needed to compute all instructions in a computation block node, b) the used information is nowhere else changed during the same computation step in the whole code and c) that only information is transferred not affected of computations during this time-step are possible.

Figure 10 shows a block diagram schematically illustrating how each row of the computational and transfer matrix together, represents a combination of a chain of instructions and communication entries for one unit. Units are depending on the level of implementation (e.g. bare assembly, threads, processes, compute-nodes, etc.). Obviously empty computation cells or not used communication entries (empty cells in the computation and / or transfer matrices) vanish, as well as start- and end-communication link together as can be seen in figure 10.

Figure 11 shows a block diagram schematically illustrating one possible optimization by the inventive system providing an optimized and ideal hardware to a technical computing and processing problem. The most obvious way for such an optimization is by building different combination of rows from the computation and the transfer matrices and each combination is one possible new, parallel/concurrent code (as the transfer on same units vanish and the computations sum up) and then assess its properties on the target infrastructure, as shown by figure 11 for a simple example.

Figure 12 shows a block diagram schematically illustrating an exemplary multiprocessor system 2 having two or more processing units 21 (multiple processors) each sharing main memory 22/221/222/223 and peripherals 25, in order to simultaneously process program codes 3.

Figure 13 shows a block diagram schematically illustrating an exemplary embodiment variant according to the present invention with the source code 31 as input code to the auto-parallelizing compiler system 1 comprising the parser 11, the computational block chains module 14, the matrix builder 15, the optimizer 16, the code generator 17, the auto-parallelizing compiler system 1 generating the parallelized and optimized target code 32 as machine or assembler code for execution by the parallel processing system 2 having a plurality of processing units 21 according.

Figure 14 shows a block diagram schematically illustrating an exemplary compiler system as known by the prior art systems, in comparison to the inventive auto-parallelizing compiler system 1.

Figure 15 shows a block diagram schematically illustrating exemplary basic elements for a tree structure and/or computation graph, respectively. The computations are reduced to one type of a statement: result = param1 operation param2. The numerical operands are +, -, *, / -> from a numerical point of view these are the only ones. Within source code 31 as high level language, values are assigned to variable in the form of variablename = value. This is a link to a „virtual" place (at variablename) to save a value. The source code 31 is normally sequential – this is how a programmer thinks. According to the invention, the code is differently approach seeing the code as $savelocation_1$ = $savelocation_2$ operation $savelocation_3$. Based on this, basic graph or tree structure elements are here introduced with e.g. two input data nodes 33/331/3311 ($datanode_{in1}$ and $datanode_{in2}$), an operation node 332 and an output data node 33/332/3312 ($datanode_{out}$). They are connected with directed edges as seen in figure 15. A node 33, as used herein, is a basic unit of a data structure, such as a sequence of linked processor instructions (computation block node 333/operation node 332) or tree data structure, as a data entry or data output structure ($datanode_{in}$/$datanode_{out}$). Nodes 33 contain data and also may link to other nodes 33. Links between the nodes 33 are often given in the graph of figure 15 by pointers (arrows).

Figure 16 shows a block diagram schematically illustrating exemplary the basic elements of the source code. In the lexer and parser 11, operations from the higher language are reduced to the basic graph or tree structure elements. So it becomes possible to use: (i) arithmetic operations: +, -, *, /, (ii) logical expression like a condition: a == b or a >= b or similar leading to a true or false output, (iii) variable assignment name = value, (iv) flow control operations as (1) branch expression such as if (condition) {block1}, and (2) loop expression like loop (condition: incremental variable: maxvalue for incremental variable) {block1}. Here a similarity exists to the basic operation in assembler code, the instruction sets in a processor 21, which have normally 3 parameters. A statement and a basic graph element describe two read ($datanode_{in1}$ 3311 and $datanode_{in2}$ 3312) and one write access ($datanode_{out}$ 3312) after an operation (operation node 332). Multiple assignments to a variable name

create versions of this variable and create different data nodes 331. As figure 17 shows, each version builds a new data node 331.

Figure 18 shows a block diagram schematically illustrating exemplary simple sequence of operations 332 in a tree structure or graph. Statement-by-statement, the sequence of the program code 31 can be added to the tree structure or graph. Complex expressions are reduced (by the need to reduce computation on a binary computing system to such a form) to a series of expression of type result = param1 operand param2. If an operation node 332 accesses a data node 331 to which another operation node 332 is writing, then the building a sequence is started by connecting these operation nodes 332 by a directed edge (next-edge). *Write* and *reads* are mapped by this method in the tree structure or graph for all data nodes 331 (see figure 19). As a rule to simplify the following steps, only 1 write is allowed to a data node 331 (this helps to simplify later operation to the graph). Otherwise a new operation 332 is introduced to model several writes to a data node 331.

Figure 20 shows a block diagram schematically illustrating exemplary how $\Delta t_{readwrite}$ 35/351 describes the time, which lies between a *write* access and a *read* access to a data node 331. With this interpretation, the latency time $\Delta t_{readwrite}$ 351 can be viewed as a number which expresses the time the data node 331 has to be transferred to another process in the multi-processing system 2. The interpretation of $\Delta t_{readwrite}$ 351 depends on the level of the hardware setup. It is the technical and physical based time between a *write* access 3311 (I/0 process, network card or similar) and the *read* access. $\Delta t_{readwrite}$ 351 is a unit given by the hardware infrastructure. $\Delta t_{readwrite}$ 351 can be composed of different Δt's: $\Delta t_{total} = \Delta t_1 + \Delta t_2 + \Delta t_3 + \Delta t_n$ (e.g. L1-cache access - memory access). For a 1-dependency (operation node read from 1 dependent $datanode_{in}$) $\Delta t_{readwrite}$ 351 is given, respectively must be very short. It is to be noted, that other problems come up, if there are 2-dependencies, which is discussed below.

Figure 21 shows a block diagram schematically illustrating exemplary the dependencies for basic tree structure elements and graph elements, respectively. As a variant, it makes it easier if the rule is set, that only one operation node 332 writes to one data node 331. Then there exist different, but finite situations to add a new operation node 332 in a sequence: (i) 0-dependency: placement: independent, (ii) 1-dependency: placement: after the dependent operation node 332; (iii) 2-

dependency: - unclear -> $\Delta t_{readwrite}$ 351 for both data node 331 (in1 3311 and in2 3312) are dependent of the history of their previous operation nodes 332; - At this point, to find $\Delta t_{readwrite}$ 351, the history of all preceding $\Delta t_{readwrite}$ 351 of $datanode_{in1}$ 3311 and $datanode_{in2}$ 3311 must be known. This history of $\Delta t_{readwrite}$ 351 has to be placed in contrast to a $\Delta t_{latency}$ 352, which is the time which is needed to distribute an information or data point to another location through the system 2.

Figure 22 shows a block diagram schematically illustrating exemplary how data/information can be considered, e.g. it is possible to transfer data from one process to another, or in case of threading, e.g. it is possible to prevent same time writing. Moving data can be seen, for example, as elementary operations (operations K) and a transfer would be $\Delta t_{latency}$ 35 long. It is important to note that the interpretation of $\Delta t_{latency}$ 35 is strongly dependent on the hardware system and can be seen as the time to transfer data/information through the system. Further, during this time operation nodes not affected by the transfer can be computed, if the system allows this, e.g. in the case of a Direct Memory Access (DMA) network transfer. Or in case of a multithreaded system 2 (where two or more processes have access to the same data, but race conditions are critical) this $\Delta t_{latency}$ 35 can be seen as a block time, where one process reads/writes the corresponding data and no other is allowed. A transmission (see figure 23) can be seen to transfer information or data through the system 2 from a source to a target location. A transmission is not yet a clearly defined pattern (e.g. send and receive) and not given if it is needed in the final code, respectively is not vanished. This only can be done after optimization to the corresponding platform/hardware infrastructure.

Figure 23 shows a block diagram schematically illustrating exemplary computation block nodes 332/ CB1, CB2,…,CBx. Computation block nodes 333 can be introduced as follows: (i) computation block nodes 333 consist of connected operation nodes 332, (ii) operation nodes 332 are composed in one chain 34, (iii) computation block nodes 333 can be connected to next computation block nodes 333 or to control flow nodes 334, e.g. branch nodes 3341, (iv) $\Delta t_{computation}$ 352 is the time to compute all operation nodes 332 (representing instructions) in a computation block node 333, (v) it has a minimum length: $\Delta t_{latency}$ 35. During this time, it is possible to transfer data in the computation infrastructure, and (vii) for describing a communication, the transmission is introduced consisting of: (a) *source*: location in computation block node 333 (start, end), computation block node 333 id, operation node id, $datanode_{in1}$/$datanode_{in2}$

3311 or output data node 3312, (b) *target*: location in computation block node 333 (start, end), computation block node 333 id, operation node 332 id, $datanode_{in1}$/$datanode_{in2}$ 3311 or output data node 3312. Transmission must not be explicitly sending and receiving information -> computation block nodes 333 can start sending and/or receiving data information at the beginning and end of the block 333 (before and after all operation nodes 332 are computed) -> this models would typically be applied in explicit non-blocking sending and receiving communication methods (e.g. Message Passing Interface (MPI)), where it can be possible to send and receive independently of computations (e.g. by Direct Memory Access (DMA) network cards). But block sending and receive work in the same way by explicitly introducing operation nodes with a send and receive operation (and then have own $\Delta t_{readwrite}$ 351). If no communication is allowed, e.g. in a multithread approach (threads sharing same data, but time dependency only can be guaranteed by lock-mechanisms), the send and write can be interpreted as lock set and release and the $\Delta t_{latency}$ 351 has to be interpreted accordingly.

Figure 24 shows a block diagram schematically illustrating exemplary the handling of program flow control elements in a tree structure or graph. The condition nodes 335 are realized based on that (i) each condition in an if condition or loop condition creates a new "level", e.g. cond:0 -> cond:1, (ii) each condition knows how many levels between the own level and the base level of a program (condition 0) are given (e.g. by cond:0 -> cond:1 -> cond:2 which means 2 condition were computed to come to this location in code), and (iii) conditions nodes 335 are connected to branch nodes 334/3341 with same condition by a directed edge. Further the branch nodes 3341 are introduced: (i) each condition has at least one branch node 3341, (ii) each branch node 3341 is connected to computation block nodes 333 belonging to this branch (code in section of if clause). The condition results from an operation node 332 with a (logical)expression with "conditional" result such as true or false. The introduction of conditions is one way to make it possible to decide to which computation block node a new operation can be assigned in the "earliest" way (the computation block node must be in the group of the same condition level but can be in another branch node). It has to remark here, that branch nodes do not correspond with a block in a branch in the definition of "basic blocks".

Figure 25 shows a block diagram schematically illustrating exemplary the handling of "*if statements*". If-statement leads to branching of the code under one

condition (if (condition) {codeA}). As the result of the logical condition expression needed to be transferred in the system -> add a following, empty computation block node 333 after the computation block node 333 of the operation node 332 with the logical command in it -> this is the time needed to transfer the result of the comparison (meaning the condition) to other processes -> it can be used to compute other things – this computation block node 333 can be marked as a branch-transmission type (CB2). After this, CB2 333 two directed edges are added to a new branch node with new computation block nodes for code in the branch1 and a so called hidden branch node (see below) which will be used if condition is not fulfilled (but it is NOT the else clause, however, it can be the else clause). The result of the condition expression has to be transferred to the following branches -> a transmission package is added to the end of CB1. Edges from branch node and hidden branch node 3342 can be set only after parsing all the statements in the branch code section, so after the closing-statement for the if condition (see below).

Figure 26 shows a block diagram schematically illustrating exemplary the handling of nested "*if statements*". As each statement in the code is added to the graph in the sequence it is read in, the control edge connection between branch nodes 3341 and following branch nodes 3341 can only be done after reaching the closing marks for an if statement -> nested if statements. The correct condition node 335 with the correct branch node 3342 can be found by calculating the last condition level (this is decreased with a closing if statement) and by traversing the graph to the correct nodes 33.

Figure 27 shows a block diagram schematically illustrating exemplary the handling of "*loop statements*". It is to be noted, that according to the branching illustrated above for if statements, loop statements can be handled with some adaptions. Loop-statement leads to branching of the code as long as one condition is fulfilled, in this example by incrementing a variable till a certain value is reached (loop (condition, i, 2) {codeA}). This is the same as for if-statement -> add CB4 333 for transmission with type loopbranch-transmission. An increment operation node 332 is added to the variable defined in the expression. The result of the condition expression has to be transferred to the following branches -> a transmission package is added to the end of CB3 333. Edges from computation block node 333 with type transmission and the branch node 3341 (branch2) after the loop branch node 3343 (branch1) can

be set only after the closing-statement for the loop statement (see below) was reached.

Figure 28 shows a block diagram schematically illustrating exemplary the handling of nested loop statements. As each statement in the code is added to the graph in the sequence it is read in, the control edge connection between computation block node 333 with type loopbranch-transmission and following branch node 3341 can only be done, after reaching the closing marks for a loop statement -> nested loop statements. This can be done by finding lower condition levels based on the last added statement and finding the correct loopbranchtransmission node 33. The computation block node 333 with the compare and increment operation nodes 332, as well as the computation block node 333 of type loopbranch-transmission has to be reconnected to the branch node 3341 of the earlier loop. As well as the branch nodes connection after the loop have to be updated

```
branch0 | cond0
for (i=0; cond1; i++)
    {branch1 | cond1
    for (j=0; cond2; j++)
        {branch2 | cond1 -> cond2
        for (k=0; cond3; k++)
            {branch3 | cond1 -> cond2 -> cond3}
        branch4 | cond1 -> cond2
        }
    branch5 | cond1
    }
branch6 | cond0
```

In regard to variable assignment, assignments are also operation nodes, but for a=0 and a=b with only one data node (in1) instead of two (in1 and in2): (i) a = 0 (--> variablename = number): - 0-dependency and 1-dependency can happen, (ii) a = b: - a copy operation node has to be added, as you explicitly copy the data from data node to another data node with correct version number, - has to be a 1-dependency, (iii) a redefinition of a variablename leads to a new version of the variable and a new data node 331. Further, array and pointer variables can be handled in a similar way.

Figure 29 shows a block diagram schematically illustrating exemplary the handling of branching and variable versions. As the compiler sees code with a clear data centric perspective, also not explicit change to data under condition has to be incorporated. In figure 30, c is computed in branch0 under condition cond0. c is

redefined in branch1 under condition cond0 -> cond1. If compare function in CB1 is false -> the program flow goes to hidden branch node 3342 hbranch1 -> copy the value from data node @ position mmapid: 12 (data node for c = 1 + 5) to new location as if in CB3 would be true. This is important and the reason there is a hidden branch node, which is not (only) the else statement. The above must be done, as the programmer programs the data flow for the true condition, but not the false condition (is expected nothing happens), but in the data dependency graph this is not true, because during parsing the code this c=8 in the branch1 has done a new version to the variable. Further, it has to be done for every variable assignment under a condition.

Figure 30 shows a block diagram schematically illustrating exemplary the handling of branching and transmissions. In the situation, when a e.g. 1-dependency or a 2-dependency occurs under different conditions -> different branch nodes -> means as branch nodes indicating that an information transmission must have happened before this. The same is also in 2-dependency cases when the same condition, but different branch nodes. However, there are also other possibilities to solve this problem. It means it needs a data transfer to this new computation block node in the corresponding branch node. Thus, branch nodes could also be handled otherwise.

Figure 31 shows a block diagram schematically illustrating exemplary the handling of arrays and dynamic arrays. Arrays can be viewed as basevariable name and an offset in a sequence of data with the same length (e.g. in C pointer variable and an offset). So it can be handled with a two-step operation: find address / location of base variable and get the offset and then go to offset found by the variable definition (type / length, e.g. u_int8 <-> u_int32). It is worth noting, that in this special case, the operation is, de facto, no really direct CPU-based elementary instruction since those are performed by the processor, itself, by first looking up in the L1-Ln caches, etc.

Figure 32 shows a block diagram schematically illustrating exemplary split and fusion situations for computation block nodes 333. In case of a 2-dependecy, there must be a fusion situation of two computation blocks 333. In case, when two operation nodes read from the same data node -> this creates a split situation of two computation block nodes 333. For both cases by introducing a new computation block node 333 (e.g. fusion-transmission or split-transmission), to split or to (re)join two calculation chains 34 (chain of computation block nodes) is allowed by sending

transferring data/information through the system. Transmission information are added to the –transmission typed computation block nodes newly created.

Figure 33 shows a block diagram schematically illustrating exemplary how to close transmission gap in graph flow. This step can happen only after all statements from the source code were added to the graph. Then to the branch nodes is added, which are not connected by computation block nodes 333 of type –transmission an additional computation block nodes 333 of type b2b-transmission. This is needed to propagate the branch change through the system. At a later step these transmission blocks must be set to the latest block number (computation step), so the information of the branch change can be propagated through the system 2.

Figure 34 shows a block diagram schematically illustrating exemplary the introduction of a time perspective to the tree structure or graph. From the connected operation nodes 332 and the flow controlled computation block nodes 333, the tree structure or graph can be seen as chains 34 of computation blocks 333 with $\Delta t_{computation}$ 352. In every computation block node, the operation nodes are grouped, so $\Delta t_{readwrite}$ 351 is "optimal" (in the sense of locality, so copy data is slower than computing data). These sequential chains in the computation block nodes should not be disrupted, because they represent the sequential part of the code. To parallel the code, the computation block nodes need to be balanced by the time to make both incoming chains of computation block nodes (and therefor chains of operations) balanced to each other -> so in a fusion situation both incoming chains have to have the same runtime. Otherwise the overall computation stalls the time the other chain has to be computed. If in all fusion situations of a computation block nodes 333, the incoming chains (CB1 and CB2) have the same runtime and start timepoint, a most concurrent/optimized parallel code is achieved. In a split situation it is possible to start different chains at the same time and create therefore parallel computations. It has to be noted that a series of "+" and under some conditions "*" could happen in a computation block node and they could be again split to parallel computations. However, this is easy to detect during an optimization phase. Each computation block node needs $\Delta t_{compute}$ to calculate all operation nodes 332 in it. Now it is possible to try to balance out the $\Delta t_{compute}$ with the systems available $\Delta t_{latency}$ 35, because each computation block node 333 can be viewed as minimum $\Delta t_{latency}$ 35 long, as this was so introduced in the preceding steps. By assuming that every computation block node 333 has to be at least $\Delta t_{latency}$ 35 long, it is possible to transfer data in the system 2. Finite

block numbers can be introduced for each computation block node. They represent the required sequence of the computation blocks 333 of the sequential chains 34 of operation nodes 332. This generates a distinguished and well-defined grouped graph, where every group of operations has a unique number and is according to these numbers transformable to bring them in matrices. These matrices then can be used to optimize/map using e.g. real world times to a given or ideal infrastructure.

Regarding to assign the block graph or tree structure to runtime numbers the numbering can be implemented, for example, as a recursive function parsing the graph of computation block nodes and their edges, branch node by branch node. In each branch node 3341, the computation block nodes 333 are started to be numbered. Then, it is stepped recursively through the computation block node chains 341 of the branch nodes 3341. After a branch node, the maximal block number is known given a computation block node in a branch node. This is used for the next branch node. The rules are computation block node 333 are stepped to computation block node 333. Is there only one or no previous computation block -> set the block number to actual block number and increase the block number (for the next computation block node). Are there 2 preceding computation block nodes -> this is a fusion situation -> if this is the first visit to this node -> append to a local list of block numbers. Otherwise if this is the second visit -> use the highest block number (actual from the function call or the saved in the list of the node). Is there 1 following computation block node -> call the function for the computation block node (recursive approach). Are there 2 following computation block nodes -> split situation -> call for both the computation block node numbering function in a recursive way. If there are none (so no next computation block nodes) -> give back next branch nodes if there are and finish the actual recursive call. By appropriately adjusting the block numbers in the branch nodes transmission, the call graph is numbered based on the computation block nodes connection to a form of a discrete time graph, where each computation block node has a finite number, which has to be calculated during the same time period.

Figure 35 shows a block diagram schematically illustrating exemplary splitting the graph to single/"transmission coupled" call chains. At this point, it is possible to split the call graph or tree structure into single chains 34 of computation block nodes 333 , which leads to a single series of operation nodes with corresponding transmission information for each computation block node 333. Each computation block node has

a unique number. The needed transmission/communications are saved in the computation block nodes.

Figure 36 and 37 show block diagram schematically illustrating exemplary a possible process from transmission to directed communication. The direction of the communication can be dependent on the computation block node 333 numbers, as the examples of figure 36 and 37 show. It has to be made sure that all –transmission computation block nodes are at the highest+1 block number (to ensure to have enough time to propagate information of program flow to all computation units / processes) in one branch node. Transfer the transmission packages with source and target information explicitly to send commands in the correct computation block nodes. The receive part can only be done after the optimization step. Signals transmission, such as a branch- or loop-transmission has to be implemented to all connected next computation block nodes 333 in the corresponding branch.

Figure 38 shows a block diagram schematically illustrating exemplary the process from graph to matrices. At this point, the graph consists of branch nodes with single chains 34 of computation block nodes 333. Each computation block node 333 knows to which other computation block node 333 what information (data and or signal) has to be transferred. Each computation block node 333 has a discrete number, which represents the order each block has to be computed over the time of the program. So the graph can be interpreted as matrices where each cell is a computation block node, each row a single unit and each column the discrete block number (order number) and each cell (computation block node) knows what for information has to be send to other computation block nodes 333 (cell entries). Each cell knows which operations (chain of operation nodes) has to be calculated during their block number and in which order – they are independent and have to be in the correct order -> which have a clear nearest write and read connection. Each operation node knows the correct global data which is needed to read (in1 and in2) and write (out). The matrices now can be optimized, e.g. numerically, to the target hardware infrastructure (CPU, CPU with explicit network transfer e.g. by MPI, GPU (memory transfer then vectorized operations), etc.)

Figure 39 shows a block diagram schematically illustrating exemplary program paths and matrices. z-axis dimension is a code perspective for each path through the code (each condition change introduces a different path through the

code). The 3rd dimension (or z-dimension) is a condition switch meaning each set of a transfer and a computation matrix is one way through the code under 1 condition. In this perspective, each path through the code could be represented by two matrices with row for the independent chains and columns (x- and y-axis) for the block numbers. Branching evolves by switching to the corresponding z-dimension set of matrices (path through code) and each process node needs to know all possible chains 34 for itself. In this description each condition change is captured by one set of a computation and a transfer matrix, but matrices can also be seen a tensor and then e.g. a computation and a transfer tensor can embrace in their rows the independent operations, respectively corresponding transfers and in each column a discrete computing step and in their 3rd dimension different conditions. Or all could be packed in one tensor combining computations and transfers for each block number and unit and condition, etc..

Regarding optimization, the now applicable automatic optimization techniques are numerous: combine rows to reduce parallelity, move single operation chains to the earliest point (send commands in the cells are like barriers), reduce communication by best combination of rows, etc. The compiler system 1 can be used to get an estimation for runtime of a cell entry, or ways like by Agne Fog of the technical university of Denmark can be used to extract CPU and cache interactions, or use tables from the manufacturer of the CPUs, or compile it with openCL, etc. There is a wide range of optimization techniques to map the matrices to a target infrastructure in the technical fielders. For example in the perspective with one tensor for the computations and one for the transfers, different parallel code versions can be retrieved by combining the same rows in each tensor and therefore generate new combinations of "computations and transfers" per block number . This would lead to reduce the number of parallel / concurrent units. By combining rows it is possible to reducing or group transfers (in a final code communications) and the computations sum up. Operation nodes in a block can be move to a previous block in case there is no send() or read() at the beginning of the block. Each cell knows the amount of data (memory or similar) it needs. The different given $\Delta t_{latency}$ 35 on the target platform can be used to make a decision on which sequential part (cell entry in computation matrix) has to be computed at which hardware unit. The communication type in the infrastructure can be implemented as needed, from asynchrony or non-blocking to blocking explicitly send and receive commands in a MPI framework, to prevent race conditions by ensuring correct barriers are set and released, to bulk copy transfer in

GPU infrastructure, etc. The choice of the optimization technique can, thus, easily be done by choosing an appropriate prior art optimization technique, e.g. by use of a SOTA compiler for each code (computations and transfers) per unit. The used optimization techniques will introduce a much wider perspective to get a more „perfectly parallel" code than others (perfect in the mean of a linear dependency of speedup vs. # processes with a slope of 1). So it is possible to numerically optimize the matrices to a new parallelized and optimized/concurrent code for the target hardware. This can be done automatically and therefor it is a big step compared with other methods. As this is done by software, software now can parallel its own code, which is new and leads to new possibilities, for e.g. adaptive models in Machine Learning (ML) or Artificial Intelligence (AI) applications, meshes in Computational Fluid Dynamics (CFD) calculations or particle sources in vortex methods, or combining different model methods with different spatial and time resolution (Finite Volume Methods (FVM) with agent-based models and statistical models), etc.

Figure 40 shows a block diagram schematically illustrating exemplary code extraction. Code can be extracted from the matrices by a wide range of implementations: directly extract assembler code, write code files and compile them by state-of-the-art compilers, implement send and receive methods explicitly, e.g. for Message Passing Interface (MPI). The computation matrix cells have the code = statements in form of the operations, and each column has to be computed during the "same" time-intervals and each row is a separate process. The transfer matrix cells know what information has to be shared with what other cells.

Figure 41 shows a block diagram schematically illustrating the computation block nodes (CB) with the operation and corresponding data nodes (similar to the token in table 1) for a double recursive call of a function.

Figure 42 shows a block diagram schematically illustrating the recursive call of the functions resulting in additional transfers. indicates a transfer to the location in the computation block nodes (start- or end-point of the computation block nodes).

Figures 43, 44 and 45 show a block diagram schematically illustrating the step of numbering the computation block nodes depending on their call-position in the code, resulting in a pseudo graph like schematically represented in figure 43, which in

return results in the computation and transfer matrix for each path number as shown in figures 44 and 45. In the perspective to see a matrix as a m x n object, then there will be created one set of a computation and a transfer matrix per path number, resulting in 2 computation and 2 transfer matrices. The switch of the paths (or conditions) can be seen in figure 46, where the 'True'/'False' signaling is indicated.

Figure 46 shows a block diagram schematically illustrating the result of combining start- and end-communication cells in the transfer matrix and eliminating empty cells in the computation matrix (for each path) and bring them back to different code segments. Based on this code segments a code can be generated directly (as a compiler) or transferred back to code and then using a SOTA compiler to generate machine code (as a transpiler), e.g. to use the unit specific optimization techniques implemented in SOTA compilers, which target mostly on one unit compilations.

Figure 47 shows a block diagram schematically illustrating how a function call in a parsed code can be handled with the presented method. After the parsing of the function definition the resulted computation block nodes (or matrix entries) can be "copied" / used at the position, where the function is called.

Figure 48 shows a block diagram schematically illustrating how the inventive method maps and/or optimize the code in a more concurrent solution than the input code. As indicated to optimize the code it is the combination between the rows, the graph is shown with the combination of the cbn in the branch node marked as branch2b (see figure 43) and combine them, see figure 48. Calling a function is in the method placing the computation block nodes to the correct location in the matrices, respectively to apply the corresponding transfers as indicated in figure 47. With this in mind, the recursive call of the function can be seen with the inventive method as a transfer and 'read' and 'write' of the function parameters and result variables in a return statement, figure 48.

Figure 49 shows a block diagram schematically illustrating, according to the example of a function call to calculate the Fibonacci number for input 4 (fib(4)), step-by-step how additional computational block nodes and the reducing of the transfers (as all is on one computation-chain) results in a more optimized source code.

Figure 50 shows a block diagram schematically illustrating step 4 showing the cbns for a call with 'n=4', where, as recursive calls can be very simply detected in the code, it is easy to not implement a recursive call in full dimension in a final application, as shown by figure 49-53 with some simplification, where the depth of this chain depends directly on the number n in fib(n).

Figure 51 shows a block diagram schematically illustrating step 5 showing the transfers, which will be occur (transport the information of the last "write" to a data node to the place where the "read" of a data node occurs and remember this information in the corresponding computation block nodes).

Figure 52 shows a block diagram schematically illustrating, as all computations are on one chain, the transfers will vanish, as can be seen in figure 52.

Figure 53 shows a block diagram schematically illustrating that, when resolving every step, this leads to a program of the form in figure 53.

Figure 54 shows a block diagram schematically illustrating that a function call and therefore a recursive call can also be interpreted like an array operation applying the inventive method, as it is transferring an information (function parameter param[i]) to the corresponding cbns in the branch node of the function declaration and then transfer back the return value(s) to a[i].

Figures 55 and 56 show a block diagram schematically illustrating an implementation in python to solve a 2D heat equation. Each entry in an array is a data node. Reading from an array index is in the inventive method an operation node with data node of the index and the base address of the array. Array operations can be seen as operation nodes with corresponding data nodes, see figure 56.

Figure 57 shows a block diagram schematically illustrating, for the example of a loop array with an array operation $a[i+\Delta iw] = a[i]$, the corresponding computation block nodes.

Figure 58 shows a block diagram schematically illustrating the initial block (figure 55) being expressed in detail in the graph as shown in figure 58.

Figure 59 shows a block diagram schematically illustrating that by using a 1D array notation and applying the fundamental rules of the method, a model approach is derived.

Figure 60 shows a block diagram schematically illustrating that a statement such as the array operations in the j-loop (figure 55), result in 5 computation block nodes, representing "reading" operations of the array, then a computation block node which computes the arithmetic solution and then a computation block node with the write to the array at the position [k+1][i][j]. This form of representation is a schematic perspective to show more clearly how the method take such array operations into account, leading to a situation indicated in figure 60.

Figure 61 shows a block diagram schematically illustrating a scheme that can be derived from the fact that each loop creates therefore a new computation block node with a "reading" or "writing" operation node and the corresponding transfers. So each loop iteration introduces the computation block nodes and form for one unit (later unit which computes the operations) a chain of read, compute, write steps. Transfer will be introduced in case between these computation block nodes "write" and "read" dependencies occur, see figure 62.

Figure 62 shows a block diagram schematically illustrating that an offset in the index of the "reading", respectively "writing" operation can lead to a transfer between the computation block nodes containing the "reading", respectively "writing" operation nodes.

Figure 63 shows a block diagram schematically illustrating the conclusion as derived from the dependencies in "reading" and "writing" of the data nodes of ana array indicated by indices in a loop.

Figure 64 shows a block diagram schematically illustrating the dependencies between “read” and “write” operations in the "read" and "write" computation block nodes and the size of the loops (in this case the mesh size) resulting from the 2D heat equation with Central Difference Scheme with a 3 nested loops implementation as indicated in figure 55.

Figure 65 shows a block diagram schematically illustrating for one nested loop, for each loop-nest a rule can be derived to handle the gaps and the consequence to the "transfers" in the proposed model. The gaps results from looping not over the complete array / mesh / dimension, indicated with j0 and Bi in the loop definitions.

Figure 66 shows a block diagram schematically illustrating that incorporating the gaps occurring by the looping over a subset of the array, the transfer pattern of a discretized equation can be derived, and a model can be built for the transfers, depending on the gap-sizes, sizes of the loops (= in this case meshes) represented by the computation block nodes of the inventive method. The method can also be applied for each iteration for each loop and add the corresponding elements to the graph / tree structure, but this could bring some performance issues to apply the method to large problems (e.g. large meshes).

Figure 67 shows a block diagram schematically illustrating that with a very small example of nX=5 and nY=4, this leads to the set of a computation and a transfer matrix as in figure 67. In the transfer matrix the arrows '->' indicate getting information and '<-' indicating sending data to other computation block nodes.

Figure 68 shows a block diagram schematically illustrating the occurring transfers between the units, as each unit is represented by one row in the transfer matrix and the computations, represented in the computation matrix. In this illustration the entries of the computation and the transfer matrix are shown together. It is a combination of computation and transfer matrix.

Figure 69 shows a block diagram schematically illustrating that it is important to note at this point, that this does not mean it has to be sent and received, it also can be e.g. shared by a shared memory segment and guarded by a barrier or lock or vanish as it is shared by cache and therefore handled by a CPU - it depends on the used transfer / communication mechanisms. This can be transferred to a model, respectively leads to a code of the form as shown in figure 69. As the method is based on keeping all dependencies of "writing" -> "reading" and a calculation of the form a[i1] = a[i2] is first "reading" an information with a[i2] and then "writing" an information with a[i1], the operations of the first "reading" (p0 – p1) are introduced.

Figure 70 shows a block diagram schematically illustrating how the transfers can be used to create a time model. Grey are the meta values (e.g. known from variable type definitions or similar), which also can be used to map/optimize the matrices to a given hardware infrastructure. These values are not used in this example.

Figure 71 shows a block diagram schematically illustrating a simple model fully resolved to illustrate possible optimizing steps.

Figure 72 shows a block diagram schematically illustrating the behavior, which can be derived for a grid of nX=2048 and nY=1024 for Δt = computations in combined cbn * cpuPower+number of transfers*networkLatency with two different set of cpuPower and networkLatency.

Figure 73 shows a block diagram schematically illustrating that the Fibonacci source can also be implemented using loops. According to the example with the 2D heat equation, this leads to the result in figure 73.

Figure 74 shows a block diagram schematically illustrating that applying the inventive method to the technical problem of pointer disambiguation, it can be shown, that the inventive methodology resolves the disambiguation occurring by passing function parameters as pointers, because it takes the pointer as an information and this disambiguation will be resolved in the step of the transfer vanishing, as illustrated by figure 74. It is to be noted that the labels "Intel P5 Infiniband" and "Intel Haswell L3-cache" can generally denote a configuration 1 and a configuration 2.

Figure 75 shows a block diagram schematically illustrating LLVM IR and corresponding grouping of instructions in computation blocks (CBs).

Figure 76 shows a block diagram schematically illustrating the forming of compute and transfer segments.

Figure 77 shows a block diagram schematically illustrating a time model structure for compute and transfer segments for n_{unit}=3.

Figure 78 shows a block diagram schematically illustrating an option for 2 units, marked as combination (c1) and combination (c2) for n_{unit}=2.

Figure 79 shows a block diagram schematically illustrating the runtime for one unit: n_{unit}=1.

Figure 80 shows a block diagram schematically illustrating a representation of decomposed code as matrices or graph.

Figure 81 shows a block diagram schematically illustrating the forming of a gamma-graph for fib(3).

Figure 82 shows a block diagram schematically illustrating a computation graph with serial and parallel time estimations.

Figure 83 shows a block diagram schematically illustrating three different computation graphs for TT = 1, 3, 10.

Figure 84 shows a block diagram schematically illustrating an embodiment variant using the system and method is in the middle end of an auto-parallelization compiler, i.e. the use of method in SOTA compiler.

Figure 85 shows a block diagram schematically illustrating a representation of decomposed code as matrices or graph.

Figure 86 shows a block diagram schematically illustrating combining of CBs and resulting distributing to computing units.

Figure 87 shows a block diagram schematically illustrating distinguished computing and transfer/communication sections from a gamma-graph.

Figure 88 shows a block diagram schematically illustrating paths in a code in basic block (BB) and control flow graphs (CFD) representation and with the inventive computation block (CBs).

Figure 89 shows a block diagram schematically illustrating a graph with computation blocks with one if-condition leading to a branching.

Figure 90 shows a block diagram schematically illustrating computation and communication segments for 2 units and one if-branching.

Figure 91 shows a block diagram schematically illustrating a loop-body in Control Flow Graph as *read*, *compute,* and *write* computation blocks (CBs).

Figure 92 shows a block diagram schematically illustrating an unrolled representation of computation blocks and transfers for i=0-7.

Figure 93 shows a block diagram schematically illustrating loop-sections with parallel CBs and explicit loop iterations.

Figure 94 shows a block diagram schematically illustrating a loop-section data analytic 'A - resolved' or 'B - structure' (B-model). Getting gamma-nodes from data analysis in a loop is a NP-complete problem (A - resolved). With the introduced embodiment variante of the method it is possible to get the gamma-node from the loop-headers/latch and the statement(s) for accessing a random access data structure.

Figure 95 shows a block diagram schematically illustrating a generic structure with CBs for a n-nested loop with a statement to a random access data structure (e.g. an array).

Figure 96 shows a block diagram schematically illustrating case 1 with gamma-nodes for 1-read-write dependency in CB.

Figure 97 shows a block diagram schematically illustrating case 2 with K-reads and 1-write, with K=2.

Figure 98 shows a block diagram schematically illustrating case 2 with K-reads and indicated how to retrieve $n_{||}$ and K=3.

Figure 99 shows a block diagram schematically illustrating combining parallel CBs leads to reduced transfers and summed up computation on n_{Unit}=2.

Figure 100 shows a block diagram schematically illustrating building combinations of CB as a function of read-shifts leads to reduced transfers.

Figure 101 shows a block diagram schematically illustrating building combinations of CB as a function of read-shifts leads to a distinct function describing number of combination n_{comb} and size of transfers T.

Figure 102 shows a block diagram schematically illustrating the minimal transfer per CB and constant transfer $S_{T,const}$ for $n_{comb} > n_{comb,const}$. Transfers between the CB i>3 and i<blocksize-5 vanish.

Figure 103 shows a block diagram schematically illustrating optimal distributing of gamma-nodes, when resources are limited. by adding transfers to distribute Gamma-nodes evenly over limited resources.

Figure 104 shows a block diagram schematically illustrating structured data and resulting gaps when not iterated over the whole array.

Figure 105 shows a block diagram schematically illustrating gamma-nodes with CBs and transfers after each n_{loop} iteration and their representation as gamma-nodes.

Figure 106 shows a block diagram schematically illustrating different possible embodiments EV1 to EV5. EV1 of figure 106 shows the most fundamental embodiment variant enclose the basis of all other embodiment variant of the inventive system and method providing auto-parallelization of code by optimizing the overall latency time to a minimum. The embodiment variant EV1 is based on some fundamental assumptions, which are, inter alia, that (i) the optimization of the auto-parallelization is not limited by the number of parallel processors (single-core 2103 and/or multi-core 2102) available, (ii) the latency time for memory access and data transfer to and from a processor 2102/2103 to the processing time needed to compute a computation block node 333 on a processor 2102/2013 is small, and (iii) the time difference to compute different computation block node 333 is neglectable. The latter can be assumed based on the inventive choice of the extremely basic structure of the computation block node 333. To optimize to limited resources (given maximal rod-dimensionality) a reduction of the row dimensionality can be obtained by building

combinations. This form of optimization is bounded with the assumptions (i) any transfer introducing a significant latency time compared when a transfer in the transfer matrix vanishes $\Delta t_{transfer} \gg \Delta t_{transfer} \approx 0$. (ii) variations between transfers are small $\Delta t_{transfer,i} \cong \Delta t_{transfer,j}$, meaning all transfers have approximate equal latency times. Assuming (i) and (ii), the maximal row-dimension in the compute and transfer matrix define the combinatorial complexity of the system (several CB per BB). Following Wall, David W. "*Limits of instruction-level parallelism*." Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, 1991 there are on average 3 to 4 ILP opportunities per Basic Block. (i) and (ii) are given for most of modern multicore-processors. Nevertheless, these assumptions can lead to miss some optimize opportunities and to potential limits for the combinatorial optimization step, especially for loop-sections and / or if (iii) the units have not similar computing performances and transfer latencies.

EV2 in figure 106 shows an embodiment variant comprising the process of building the inventive tasks out of combination of computation block node 333 belonging (i) to the same column in the computational matrix, i.e. executable in parallel due to the available data and / or (ii) continuous computation block node 333 in the same row with empty cells in the transfer matrix for the same branch node 334 (=compute segments). This allows for a processor-architecture specific and/or system-architecture specific optimized auto-parallelization where the number of processors 2102/2103, the performance of the different processors/cores 2102/2103 and/or processor units 21, the differences in size and response time of the different memory units 22, in particular the different processor registers 2211 and/or processor caches 2212 and/or RAM units 2213. These hardware-specific structure parameters can be included in the optimization of the auto-parallelization and thus equalized by building tasks 36 of different length, e.g. in the number 36i of processed computation block nodes 333 of a task 36, where the computation block nodes 333 of a task 36 belong to the same column of a computation matrix 151 or are continuous computation block nodes 333 of the same branch node 334 without an entry in the corresponding transfer matrix 152. This differentiation is illustrated in figure 80 in gamma-graph a) and b).

EV3 in figure 106 illustrates handling parallelization of random accessed data structures in loops by the inventive system 1 and method, without explicitly resolving data-dependencies in the loop sections instead by evaluating the read-write dynamics from the affected statement(s) to access the random accessed data

structure and the influence of used loop-variables by analyzing the loop-header/leap-definition(s).

EV4 of figure 106 (more detailed illustrated in figure 110) illustrates the applications of the inventive system 1 in its use for optimization of integrated circuit (IC) or chip design solving electronics engineering technical problem if IC design by encompassing logic and circuit design required to design integrated circuits, or ICs.

EV5 in figure 106 illustrates an embodiment variant of the inventive system applied for getting quantum gates in quantum computing systems. In classical computing the basic memory element is a bit and can have 0 or 1. 0 and 1 is represented on the electronical level as two distinct voltages (see e.g. Basic quantum algorithms by R. Portugal). To get a result for a classical computation, the voltage at the output of the computation is measured as voltages. In quantum computing the units are qubits which are assumed to be at the end of a computation also 0 or 1. In contrast to bits, qubits can be simultaneous of 0 and 1, meaning coexisting during computation. The quantum coexistence (before a measurement) can be captured by a linear combination of orthonormal vectors (see e.g. Basic quantum algorithms by R. Portugal). With a measurement the quantum system gets inevitably affected and results in a stochastic result, resembling the classical bit. The definition of a unitary matrix is if its conjugate transpose UT is also its inverse UTU = UUT = UU-1=I, where I is the identity matrix. To manipulate the state of a qubit (see state as the "value" of the qubit before the measurement, quantum gates are needed where a single-qubit quantum gate can e.g. be expressed as a 2x2 unitary matrix. Any matrix describing quantum gates must be unitary, as quantum gates must be reversible and preserve the probability of the amplitudes. In the 1970s Charles Bennett showed how it is possible to transform any classical computation into a reversible form (see Logical reversibility of computation by C. Bennett). This can be done by keeping intermediate data with extra memory. Seeing f(x) as representing a classical computation, f(x) enables to manipulate n input bits to m output bits. Simplified, by keeping the garbage g(x) each classical gate can be transformed in a reversible version. Figure 113 (below) shows the classical computation for input xi to output by f(x). Such an approach enables to compose from a given classical computation circuit a reversible form. This reversible form makes it possible to retrieve quantum gates with the needed unitary properties. An illustrated form of this approach is given by 114 (below).

Following Figure 111 (below), it is possible with the inventive system to automatedly retrieve, e.g. from nested-loop and statements of accesses to an array (EV3) a representation as classical combinational integrated circuits (EV4). A perspective of the method is to see the decomposition in computation blocks as extracting a unique data bit-size for each computing step minimal needed to compute the input bits in parallel. This is illustrated and marked as "independent chains" in figure 39. Each row in the matrix represents a needed data size to compute this step in parallel in the program. This information can be used and could enable an approach to derive quantum gates also from (nested-) loops. The input bit-size per stencil can be derived by $\sum(t_i + g_k) \cdot dr$, where $t_i \, \epsilon \, [a, b, c, d, e]$ describe the input bits (see figure 113), $k \, \epsilon \, [0,6]$ are the garbage memory bits needed and dr the corresponding data-sizes (e.g. 32 or 64 bit for float values). The method enables the extraction of the parallel computing blocks, also for complex cases as an implementation of a 2d heat equation implementation as in Fig. 55 shown. EV3 enables the extraction of parallel computing blocks without resolving all data dependencies and retrieves the number of parallel computation blocks in the loop $n_{\|} = n_{isub} \cdot n_{jsub}$, see figure 64. This data size information enables to retrieve the needed number of qubits as a function of runtime-time parameters (n_{isub} and n_{jsub} are a function of nX and nY = mesh, respectively problem size). The computational base for quantum computations is defined by 2^n, where n is the number of qubits. As the method enables to extract from the parallel computation blocks parallel combinational circuits (figure 111g) the method is able to derive unitary matrices (quantum gates) to compute a set of input bits as in figure 114 shown with f(x).

Figure 107 shows a block diagram schematically illustrating the embodiment variant EV2 of figure 106 more detailed. The embodiment variant EV2 comprises the process of building the gamma graph out of the computation block node 333 of EV 1, wherein the computation block node 333 and gamma-node in the initial granularity composed of the cell in the computational matrix. Gamma-nodes connected with one edge can be combined in a later step to one gamma-node, as they consist of one compute segment. A compute segment is a series of CBs of the same branch node 334 without any transfers in the transfer matrix. This allows e.g. for a processor-architecture specific and/or system-architecture specific optimized auto-parallelization in respect to the number of processors 2102/2103, the performance of the different processors/cores 2102/2103 and/or processor units 21, the differences in size and response time of the different memory units 22, in particular the different processor registers 2211 and/or processor caches 2212 and/or RAM units 2213. In the

representation of the gamma-graph the edges represent transfers $\Delta t_{transer}$. This enables new, bounded optimizing approaches, e.g. to reform a gamma-node by fixing all transfers $\Delta t_{transer}$ by a hardware defined Transfer Times (TT). Traversing the graph it is feasible to define for a parent node by combining child nodes (runnable in parallel) if it is beneficial to compute the child nodes in parallel and adding the transfer overhead TT or run all or some of the child node in serial (e.g. in a multithreading environment can TT be seen as the context switch time) without having the overhead of parallelizing.

Figure 108 shows a block diagram schematically illustrating the embodiment variant EV3 of figure 106 more detailed by illustrating the applications of the inventive system 1 to loop parallelization. This can be illustrated along the 2d heat equation example, where a gamma-graph can be derived as a function of the read-write distance ΔI_{rw} obtained by the array read and write statements formed by accessing the array u[i+nX*(j*nY*k)] in the loop-body. The change of the loop-variable i, j and k and their dependency on the runtime-variables can be obtained from the loop-header/latch. It is to be noted that since parallelization, i.e. optimized generation of multi-core or multi-processor code (in particular the task scheduling) is considered of most of the codes as NP-complete problem, the predominant opinion in the prior art is that optimized parallelization can only by achieved applying heuristics. The method with EV3 allows for loop-sections with statements including random access data structures in conjunction with loop-variables, a non NP-complete solution. Normally, loop-sections are where in codes the highest computational effort emerge. The inventive system EV3 and method allow to get a simple, non NP-complete system for loop-sections. Studies performed on a running system of the present invention proved by starting from discretized 2D heat equation (PDE) source code to provide optimized executable Message Passing Interface (MPI) parallel code (send and receive principle, i.e. data dependence must be fully resolved), where MPI is a portable message-passing standard designed to function on parallel computing architectures. Additionally, EV3 and method allow to reduce the complexity in finding optimal solution with exact or heuristic methods in EV1 and 2, see Fig. 109b.

Figure 109a shows a block diagram schematically illustrating the embodiment variant EV2 of figure 106 more detailed by illustrating the applications of the inventive system 1 to generate a gamma-graph, which can describe a Mixed Linear Integer Programming problem (MLIP). A general integer linear programming (ILP) problem is defined as:

z_{ILP}^{*}: = minimum $z(\boldsymbol{x})$: = $\boldsymbol{c}^{T}\boldsymbol{x}$

$\boldsymbol{Ax} \leq \boldsymbol{b}$ und $x \in \mathbb{Z}_{+}^{n}$

where z: = $\mathbb{Z}^{n} \rightarrow \mathbb{Z}$ denotes the objective function with its set of feasible solutions by S_{ILP}: = $\{\boldsymbol{x} \in \mathbb{Z}_{+}^{\boldsymbol{n}}: \boldsymbol{Ax} \leq \boldsymbol{b}\}$. As an example following the work described in R. Salman, "Algorithms for the Precedence Constrained Generalized Travelling Salesperson Problem", Chalmers University of Technology, University of Gothenburg, Sweden 2015, the problem of optimizing the gamma-graph can be formulated as a MILP problem such as the proposed Precedence Constrained Generalized Travelling Salesperson Problem (PCGTSP). With such a set of equation, the optimization on one unit can be optimized or at least approximated with heuristic methods. Extending the problem to a parallel machine with m units, the optimization method can be used, as e.g. disclosed by R. Salman, "*Optimizing and Approximating Algorithms for the Single and Multiple Agent Precedence Constrained Generalized Traveling Salesman Problem*", Chalmers University of Technology, University of Gothenburg, Sweden 2015 in paper III F. Ekstedt, et al., "*A Hybridized Ant Colony System Approach to the Precedence Constrained Generalized Multiple Traveling Salesman Problem*", 2017. This document is explicitly incorporated by reference herein, as a whole. In particular, the system of equations and objective function with reference on page 19 and 20 (5.3a) – (5.3m), disclosed in this document can e.g. be used to generate a gamma-graph solving the discussed optimization problem of the computational and transfer matrices. Finally, this leads directly to a definition of a Precedence Constrained Generalized Multiple Traveling Salesman Problem (PCGmTSP). As the optimization step can be done during compile-time, the computational effort to cover the generic situation of code and heterogeneous hardware granularity can be (very) high from an application perspective.

Figure 109b shows how EV2 can be combined in EV3. Each loop-section can be condensed and approximated to a Δt_{loop} time as a function of the properties in EV3 n_{loop}, $n_{\|}$ which can be expressed as a function of loop-variables: Δt_{loop} = $f(n_{loop}, n_{\|}, n_{units})$. This enables e.g. to generate MILP based on two different simplification:

I. depending on the number of parallel groups in the gamma-node, the amount of n_{units} assigned to a loop-section is defined and optimized.

II. all units are used to compute loop-sections, which separates code between different loop-sections to be optimized separately, what reduces the overall complexity.

Figure 110 shows a block diagram schematically illustrating the embodiment variant EV4 of figure 106 more detailed by illustrating the applications of the inventive system 1 in its use for optimization of integrated circuit (IC) or chip design solving electronics engineering technical problem if IC design by encompassing logic and circuit design required to design integrated circuits, or ICs.

Figures 111a to 111h show block diagrams schematically illustrating the inventive system applied for its usage for IC (Integrated Circuit) design. The inventive system 1 can be applied (see figure 106/110) for both designing and optimizing VLSI designs. Circuit design is about arranging transistors to perform a particular logic function. From the design, the delay and power can be estimated. Every circuit can be represented as a schematic or in textual form as a netlist. Simply illustrated (see figure 111a), digital logic can be divided into combinational circuits (Boolean logic), whose outputs depend only on the present inputs (series of logic gates), and sequential circuits, whose building blocks are registers (flip-flops) and latches. The Inventive System 1 can be used to determine the computation block nodes. The computation block nodes in turn can be used to determine the digital logic, which will be illustrated by the example of the 2D heat equation. Fig. 67 shows that a computation block node for this algorithm consists of 5 reads, 1 compute and 1 write (figure 111b). The result of a simple circuit is shown in Fig. 111c. The reads become the sequential circuits consisting of 5 flip-flops (together they form the register, though, the effective register size is determined by the bit length of the individual data). It is assumed that a rising-edge of the clock (time k), wait for the required hold-time to be sure that the correct values (logical 0 or 1) for u[k][i+1][j], u[k][i-1][j], u[k][i][j+1], u[k][i][j-1] and u[k][i][j] are present at the output of the flip-flops. This is basically our read. The data can now propagate through the combinational circuit (computational block), which results from the arithmetic consisting of the required adder, shifter (multiplication by 4), subtractor (inverter and adder combined) and multiplier. At the output of the computational block the value

u[k+1] arises (equivalent to the write operation) after the setup time, which is the amount of time required for the input of a flip-flop to be stable before a next clock edge. The register on the right side is nothing else than the value at the next time k+1, which is needed for the next iteration on the left side. From this results that the values can be written directly into the same register. This thought experiment can now be carried out for each point on the computation matrix. Already for two points on the matrix (Fig. 111d) it becomes obvious that data are now written crosswise or that only one register per matrix point is needed. The computational block (CB) always consists of the same combinational circuit, although in the electronic implementation the propagation delays will never be the same. It is therefore important to keep an eye on the critical paths that limit the operating speed of the system and require attention to timing details. Real world settings are always limited in space with VLSI design. An infinite parallizability is not possible, so certain (not all) computational blocks are combined and processed sequentially. To make this possible for a finite number of registers, a multiplexer is connected between the output of the registers and the input to the CB. A multiplexer chooses the output from among several inputs based on a select signal. At the same time, a demultiplexer is connected between the output of the CB and the register input (see Fig. 111e) to feed the data back correctly, whereby the design optimization process might show that the demultiplexer is not always need. What is processed per clock rate on the ideal, parallel circuit is now processed in several clock cycles in the more realistic circuit (Fig. 111f/g) – as much parallel as possible in as many cycles until a complete iteration is completed. Using the 4x5 matrix from Fig. 67 as an example, this would mean: With 6 parallel CBs: 1 cycle to calculate k+1, with 3 parallel CBS: 2 cycles (2x3 parallel) including a higher delay do to multiplexing. VLSI designers always must make trade-offs between area, throughput, latency, power consumption and energy of executing a task. The optimal circuit is always located somewhere on an inverse curve (see Fig. 111h). If there is a lot of area available, parallelization can be done well. If there is less area available, the system must multiplex, which leads to a higher delay. Somewhere in between lies the optimal point between area and number of parallel CBs. Since the inventive system 1 can give the optimal code for a given unit, it can also be used to find the optimal circuit for a given area, helping in the iterative design process refined as actual module sizes and critical paths become known.

Figure 112 shows a block diagram schematically illustrating the effect of combination of CBs and the influence on $S_{compute}$, which is the (optimized = minimized)

size needed to compute a series of instructions on a target platform (e.g. register). It has to be noted, that $S_{compute\ \Gamma_1+\Gamma_2} \neq S_{compute\ \Gamma_1} + S_{compute\ \Gamma_2}$. $S_{compute\ \Gamma_1+\Gamma_2}$ as $S_{compute\ \Gamma_1+\Gamma_2}$ depends on the instructions in CB1, CB2 and CB3, target architecture (available instruction sets and data space characteristics, e.g. registers in a CPU or flip-lop / input signal in an IC-circuit) and used method to ILP optimization and scheduling.

Figure 113 exemplarily shows the classical computation for input xi to output by f(x).

Figure 114 illustrates the application of f(x) to input bits and keeping the 'garbage' values during computing the result of f(x) enables to uncompute f(x) = reversible form.

Figure 115 illustrates the application from statement in code to a combinational circuit and resulting reversible forms.

Definitions

(i) The "computation block nodes"

The definition of the term "computation block nodes", which group the instructions and entail the communication/transfer data to other computation block nodes, is crucial for the present application. The term "computation block node", as used herein, differ from similar terms, used in the state of the art, though there is no generally recognized meaning.

The well-known basic blocks (e.g. see Proceedings of a symposium on Compiler optimization; July 1970 Pages 1–19 https://doi.org/10.1145/800028.808479) are a central definition in classical Control Flow Graphs (CFG). Simplified, they group statements, which have no jump or jump targets inside. Therefore, with a given input they can perform operations without interruption to the end, respectively to the output. This is a basic concept in compilers today. The definition for basic blocks is also historically targeted for single computation units and very well established. There exist optimization methods for a wide range of problems and it has been shown how they

solve different technical problems. But seeing a code with the goal to split the statements to different dependent units (connected e.g. by a shared cache, via bus or network, etc.), this definition lacks in granularity and the classical scope prevent a broader perspective. Scoping the blocks on the bases of any unique information given in a code (see information as a bit-pattern) and combine this scope with the relevant times to compute and transfer of an information in a system, creates a different, but well physical based perspective on a given code. The alternative scope enables new option as well as solves some well-known technical problems of today's SOTA compilers (see following example for a PDE, Fibonacci or pointer disambiguations).

The term "computation block nodes", used in the present application, bases on a different and not in this way applied important unit in SOTA compilers: the interrelation of transfer and compute time for a given set of statements. These new defined "computation block nodes" group instructions together which use the same information, which is not changed by any other instructions (statements) in any other computation block node during a particular time in the complete code. In this way, they group instruction which can be processed or computed independently of any other statements with the scope on the information (information as a distinct bit-pattern), what is called in the present application "not-further splitable instructions chains". These chains of instructions in every computation block node have each a physical based "time" associated how long it takes a unit to process or compute them on a given hardware. As the hardware properties have a fundamental influence (as well as software components, such as OS, drivers, etc.) on the time needed to process or compute the instructions, the "computation block nodes" correlate them also to the time needed to possible transfers of any other information during a particular time - if needed during a particular program step - to another "computation block node" in the complete code. Each "computation block node" knows which and when the information for the own instructions has to be exchanged (communicated / transferred, respectively "received " or "send") to other "computation block nodes". Therefore, a "computation block node", as used in this application, brings a new scope and a decision criterion, meaning one of the central aspects of parallelism of a code, in relation: compute an information (bit-pattern) on a unit or transfer this information to another unit and compute it parallel. This decision can only be done, when it is guaranteed, that the used information is not changed during a particular program step (or time) in any other part of the program. Furthermore, building block nodes with this scope do not only bring advantages to parallelize a given code, this scope also shows

some advantages for problems not well handled with SOTA compiler optimization techniques. Such problems and different solutions are well documented, for example, in the publication *Modern compiler design* by D. Grune. To show the technical benefits, some advantages resulting from using the new scope to some of these known technical problems, is shown below, such as pointer disambiguation and different performances for Fibonacci series codes, but also how the method solves up till now not solvable problems, like PDE parallelization.

First, the different scope is illustrated with a schematic example in figure 6. The illustration makes some adaption to show easier, what the inventive system and method does, such as the statement 'y > a' would not occur twice in the same computation chain, the resulting "compute" -> "communicate" model would lead to a corresponding placing of a loop, jump or flow-control instruction. Nevertheless the example illustrates, what the different scope of the computation block nodes give compared to basic blocks: the fact that 'a' and 'b' do not change in block 1 and the information of the condition of 'y > a' is already known after the evaluation of the statement 'y := a*b', the alternative scope of the computation block nodes takes these properties into account. The proposed perspective of computation block nodes groups the statements in a new way together. This change in perspective results from the approach to group operations together, which rest on information, which is not changed at the same time-step in the code. In the example this is shown that two independent computation-chains evolve, which both are independent concerning the information of 'y'.

In the simplified, but more realistic example of figure 7, the inventive method exploits the fact, that a logarithmic instruction (as an example on a modern CPU: FYL2X) needs much longer than a floating point addition and/or multiplication instruction (e.g. FADD, FMUL). The inventive methods add the statement 'x:=a+b' und 'y:=a*b' to two different computation block nodes (cbn), as both are based on the same information 'a', and 'b'. The 'log2(x)' statement is appended to the 'x = a+b' as it uses the information 'x'. The information 'y' is getting transferred, so the information is available to both, independent computation-chains.

(ii) The "computation matrices" and "transfer matrices"

The inventive method forms out of such a flow-graph of computation block nodes, as discussed above, two technically defined matrices called "*computation matrix*" and "*transfer matrix*" in this application. Both are numerical matrices. The "*computation matrix*" contains the instruction chains and the "*transfer matrix*" the possible transfer properties (from and to other computation block nodes). Therefore, the code extracted from the matrices always forms a pattern "compute -> communicate", as it is described in more detail in the following passages. If code is mapped to one unit, the communicate part would vanish and the inventive method would be reduced to the approach with basic blocks, respectively can be handled with a SOTA compiler. See the example from figure 7 in this form, it can be seen, that unit 2 can compute the long running instruction of the statement 'log2(y)' and unit 1 processes the loop with incrementing a ('a = a+1'), see figure 8. To see how this works, in the next passages the matrix builder is shown in detail.

The name "matrices" is used to name a structure of the from (m x n x p), where m, n, p $\in$ $\mathbb{N}_0$. As m, n and p depend on the code, this can include different forms of a mathematical object, especially concerning dimensions like as a point, vector, a matrix, a tensor, etc.. m is the number of the maximal number of computation blocks, respectively block numbers like in Figure 43 the segment-numbers, n is the number of independent, but with the same segment number computation block nodes, like in Figure 43 indicated as chain numbers. Depending on the maximal level of conditions in a code (or series of branch nodes like in Figure 26), p is defined, indicated in Figure 43 as path number. Therefore it could be said, that whenever the term 'matrix' / 'matrices' is used, it can be a vector, a matrix or a tensor or any other object of the form (m x n x p) or with representation in from of a graph or tree. Therefore, optimization can also be done e.g. with one tensor by combining the computation and transfer matrices into one structure, or that a transfer and computation matrix each can be a tensor, or they both can be a vector, in a code with only one block / segment number. The dimension of these "matrices" depend on the form of the code and the way of handling / representing the information in the way the method is applied. The term "numerical matrices", as used herein, can also include forms of text, like for a transfer '1->2'. Depending on the used optimizing / mapping techniques, the text in the matrices will be or can be reduced to a numerical value (depending on the used character encoding) so they can be e.g. searched or compared. Or the textual transfer '1->2' can be represented / encoded by a numerical value from the beginning and directly be compared with other transfers and therefore omitting character encoding.

As the "numerical matrices" can be used as another well-formatted form of representing a graph / tree alike structure, it is also possible to work without the "numerical matrices" and do all optimization / mapping in graph / tree form. Whatever form of mathematical or computational form of representing the groups of instructions (here named computation block nodes) and their transfer dynamic represented here with the e.g. in Figure 23 mentioned "transmission packages", based on the physical based dependency of transfer and computing latency occurring in all binary based electronic computing systems. It is based on grouping instructions (any form of instruction / operation, e.g. representing a form of electronic circuit) in a code (any form of code (high level, assembly, etc.)) on the rule to place an instruction "reading" an information A in the same group (here named with computation block node) where the instruction is located which "writes" the information A. This is significant different to group instructions with the well-known basic blocks. If the dependency is not clear (indicated in Figure 21 in the case of graph elements with 2 "reading" and one "writing" node per instruction) a "transfer" is introduced between the two computation block nodes (on holding the instruction, which "writes" to one holding the instruction which "reads"). In this form the code can be represented in form of computation blocks nodes connected by their needed transfers. This information can be enclosed in graph / tree or array or any structure suitable. To run these instructions on a given hardware the well-known methods in SOTA compilers can, but must not, be used to optimize the chain of the computation block nodes grouping instructions and for each unit then to run on the involved computing units (would in this case called transpiling).

(iii) The "matrix builder" and the way back to code

After parsing the code and adding all instructions to computation block nodes (cbns), each computation block node can be enumerated depending on its position in the flow graph. This leads to a similar form of a Control Flow Graph, given by the edges between the cbns as well as the connection of the defined branch nodes. Using these positioning numbers, the unique positions in the flow of the code is used to place the information of what to compute and what to transfer in the two matrices "*computational matrix*" and "*transfer matrix*", the "*computational matrix*" for the computation and the "*transfer matrix*" for the transfers. It is obvious that meta data, such

as size of the data needed for each cbn, the size of the transfer between the cbn, etc. can easily be derived.

Matrices represent a much more scalable form of accessing information than graphs, respective show the well-formed nature of the control flow graph. They are not absolutely essential for the method and this step could also be performed directly on the graph / tree structure. But the definition of the block nodes also indicates the generic nature of the inventive matrices: Each row has an independent flow (dependency by transfers) of instructions (= computations) and needed transfers (= communications) to other cbns. It is guaranteed that a) no further information is needed to compute all instructions in a computation block node (this is similar to a basic block, but the scope of the independencies is quite different), b) the used information is nowhere else changed during the same computation step in the whole code and c) that only information is transferred not affected of computations during this time-step are possible. With this fact, each cell in the computation matrix has all instructions, which can be independently computed concurrent with all instructions in the other cells in the same column. In the transfer matrix in each cell, the needed transfers at the beginning and at the end of each computation step (corresponding cell in the computation matrix) is now known. Getting back code to run on the different units results in a representation in the form "communicate -> compute -> communicate -> compute" and so on. Each row represents a chain of computation and communication properties forming a series of computations coupled by communication with the other rows = chains of computations. The needed information to communicate with the other rows = chains is in the transfer matrix. Each row in the computation matrices (and the same combination in the transfer matrix) can also be combined (compute all instructions of both, combined cells and make the necessary transfers of both cells based on the transfer matrix) with any other row in the matrices to create a new combination of computation <-> transfer behavior of a given code. In this step the computations sum up and transfers on the same unit (by combination) vanish. This will lead later to a simple form of optimization, as well as the fact, that the optimization / mapping step results definitely in a runnable code, as no iterative or similar solution methods are needed. This is also a technical well known-problem parallelizing code, as debugging of parallel code is very complex (for example, see "*ParaVis: A Library for Visualizing and Debugging Parallel Applications*", A. Danner et al.).

Each row of the computation matrices, as defined herein, represents a chain of instructions for one unit. Units are depending on the level of implementation (e.g. bare assembly, threads, processes, compute-nodes, etc.). Obviously empty blocks (empty cells in the computation matrix) or not used communication entries (empty cells or transfer on same units in the transfer matrices) vanish, as well as start- and end-communication link together as can be seen in figure 10. This results in case there is no transfer if the code is run on a single unit and all transfers will vanish by reassigning / renaming of variables, respectively applying well known optimizing methods in SOTA compilers to get an optimized code for a given unit.

Depending on the method to implement the communication, non-blocking or blocking mechanisms can be used, as it is guaranteed, that during a computation block node no information will be transferred used in the instructions at the same time, respectively another cbn with same number. Depending on the level the computation and communication part can be implemented, resulting to use the method as a compiler or transpiler. The transfer back to code of the form "computation -> communication" approach makes it also easy to use languages (e.g. C, C++, python) suiting best for an application and depending on the target infrastructure, such as IPC (InterProcess Communication) methods (e.g. queue / pipe, shared memory, MPI (Message Passing Interface), etc.), libraries (e.g. eventlibs, multiprocessinglibs, etc.) and the use of available SOTA compilers, etc..

(iv) "Optimizations"

The generic, well defined nature of the matrices is the unique base for a wide range of possibilities to map/optimize a code to a given hardware or evaluate an optimal hardware configuration for a given code (e.g. see the different embodiment variants E1-E5 in figure 106). The structure guarantees as a result a runnable code. Each hardware infrastructure has its own performance properties and in combination with the available software layer, a modern ICT (Information and Communications Technology) infrastructure is very complex. The inventive method allows to optimize the code to the hardware or allows to give out an ideal hardware. The most obvious is by building different combination of rows from the computation and the transfer matrix, whereby it is important that in both matrices the same combinations are built. Each

combination (e.g. combine row 1 and 2 in computation AND transfer matrix) is then a new version of a parallel / concurrent code for the given input code and then its properties on the target infrastructure, see figure 11 for a simple example, can be assessed.

By different combinations of the rows in the matrices (e.g. combine row 1 and 2 in computation and transfer matrix or row 2 and 5) different combinations of computation <-> communication ratios are retrieved. Each of the combinations then can be examined including other known meta-data for a given hardware / software infrastructure. For example data types of the data nodes () can be used to evaluate for hardware properties, such as cache-lengths, available memory or other properties of a target platform. This form of combining and searching for an optimal compute / communication ratio for a given code and a given hardware results always in a runnable, concurrent code, because no iterative solution method or similar approach is needed to find a solution in the optimizing step, nor solution with e.g. race-condition, dead-lock, etc. can occur, respectively dead-locks can be detected.

The grouping of the instruction is based on the inherent physical constraint that transfer is normally a magnitude larger than to compute at the same "location", the method spawns a form of optimal solution space for the most splitable form of a code and results in a well-defined way and search space for a unique way to find an optimal map for a given hardware, or the ideal hardware for a given code. This makes this method quite generic and solves the technical problem to adapt a given code to a target platform automatically. In case the method is used as a transpiler, SOTA compiler could be used to optimize then the code to a target hardware / unit.

Other methods do not exploit the inherent time properties and dependencies given in a code and do not produce this form of unique solution space to optimize, in the form the method's definition of the computation block nodes (by grouping by the scope of no changing information) - this solves technical problems directly in many ways. The examples in the detailed description will show this in more detail.

(v) "Computation block nodes" according to the invention and "basic blocks or slices" of the prior art

Computation block nodes are NOT the same as basic blocks or slices: They have a different scope. They do not follow the definition of basic blocks, e.g. by connecting independent program parts based on their jump / branching. Instruction chains in computation block nodes have an elementary data dependency, meaning that the information used in these chains is nowhere else in a given code changed nor transported at the same time/program step.

So the time, respectively the place of an instruction in a program is brought into dependency in the computation block nodes. The computation block nodes consist of a chain of instruction, which is based on the same information, whereby information is defined as every form of bit-pattern (e.g. data variable, a pointer-address, etc.). By introducing a transfer/communication, where an information is not only used by a time-step in a code and coupled with the instruction, a correlation of these both physical based times are reached in a computation block node. The method places the computation blocks in such a way, that to all timepoints is definite which information can be transferred and where to and which information can be computed in parallel. This gives a different perspective, especially for optimization techniques. A wide range of technical problems exists, which can be solved by this change in perspective. The computation block nodes connect the location of an information (bit-pattern) in a computation framework with the time this information is used in a program. This is backed by the elementary physical principles of computing in a classical infrastructure. The following example with nested loops and arrays shows this effect good: the branches in the loop definitions can be translated to a read and write of data points with according to arrays when distributed over parallel computation block nodes.

The inventive method splits a code into segments of computation and where the resulted information has to be transported. Therefore, by the definition of the computation block nodes the inventive method generates a matrix system where it is given, that instructions are grouped which base on the same information and the constraint that during any given timepoints nowhere else the same information is needed in the infrastructure. The grouped instructions are not further splitable, because as it is not possible to reach a faster computation for a particular group of instructions,

because the transport in any form would be longer than to compute this given chain in the computation block nodes. As shown the generic nature of the computation and transfer matrices makes it possible to optimize the split code to a most concurrent as possible solution for a given hardware. The ratios of computations and transports are depending on the target hardware and the method provides different solutions to split to different ratios for a given code. By translating the graph of dependencies into matrices these can be used in a more effective way to map / optimize the split code to a target platform, including the specific properties of this infrastructure (e.g. GPU need another handling of transfer / computation distribution, than CPUs). But it is not essential to use the matrices and the optimization / mapping could be done directly on the graph / tree structure.

(vi) "idle time" – "latency time"

The latency times, as defined herein, is given by the idle time of a processing units 21 between transmitting data back to the parallel processing system 2 after having processed a specific block of instructions of the processing code 32 on data by a processing unit 21and receiving data (i.e. after retrieving and/or fetching the data) necessary for execution of a consecutive block of instructions of the processing code 32 by the same processing units 21. In contrast, idle time of a processing unit can be defined herein as the amount of time the processing unit is not busy between two computational block nodes, or, otherwise, the amount of time it executes a system's idle process. Thus, idle time allows to measure unused capacity of a processing unit of the parallel processing system. Maximum speed-up, efficiency and throughput are the ideal cases of parallel processing but these are not achieved in the practical cases, as the speedup is limited due to various factors contributing to the idle time of a processing unit. Idle time of a processing unit, as used herein, can find its origin in various causes, inter alia, comprising: (A) *Data dependency between successive computational block nodes,* (i.e. tasks which cannot further be split up framed by read/write operation): There may be dependencies between the instructions of two computational block nodes. For example, one instruction cannot be started until the previous instruction returns the results, as both are interdependent. Another instance of data dependency will be when that both instructions try to modify the same data object, also referred to as data hazards; (B) *Resource Constraints*: When resources are

not available at the time of execution then delays are caused in pipelining. For example, if one common memory is used for both data and instructions and there is need to read/write and fetch the instruction at the same time then only one can be carried out and the other has to wait. Another example is of limited resource like execution unit, which may be busy at the required time; (C) *Branch instructions and interrupts in the program*: A program is not a straight flow of sequential instructions. There may be branch instructions that alter the normal flow of program, which can delay the execution and affect the performance. Similarly, there can be interrupts that postpones the execution of next instruction until the interrupt has been serviced. Branches and the interrupts can have damaging effects on the minimization of the idle times.

It is to be noted, that the task of minimizing idle times is also sometimes referred to as "load balancing", which denotes the goal of distributing the work among the processing units so that all processing units are kept busy in the ideal case all of the time.

(vii) "Elementary instructions"

The term "elementary operations" or "elementary instruction", as used herein, refers to machine operations that contains no simpler operations. They are on machine language level. However, in specific embodiment variant, they can at least comprise microcode instructions or completely consist of microcode instructions. As microcode is the result of lower-level machine language interpretation, where microcode instructions directly manage hardware resources at the register or circuitry level. Machine language interprets and sends machine instructions to the lowest hardware layer level where they are translated into small micro programs called micro-codes. Thus, microcode is low-level code that defines how a microprocessor should function when it executes machine-language instructions. Typically, one machine-language instruction translates into several microcode instructions.

As discussed, the execution of an elementary instruction, as defined above, consists typically in the successive execution of some number of operations, including such operations as resetting a register, resetting memory storage, shifting a character in

a register left or right by one place, and transferring data between registers as well as the comparing of data items and logical addition and multiplication. A set of elementary operations can provide a structure for executing a specific instruction. Elementary operations comprise basic logical functions of logic gates comprising AND, OR, XOR, NOT, NAND, NOR and XNOR. Such elementary operations can be assumed to take a constant amount of time on a given processing unit and may change only by a constant factor when run on a different processing unit 21 or parallel processing system 2.

Building on the concept of microcode instructions, the design of integrated circuits such as Field-Programmable Gate Arrays (FPGAs) and Application-Specific Integrated Circuits (ASICs) involves several levels of abstraction, from the high-level algorithmic description down to the low-level hardware description. These descriptions are either written in a Hardware Description Language (HDL), which allows designers to define the functionality and behavior of their circuits at a very low level, or more and more often the process of High-Level Synthesis (HLS) is applied to describe the behavior of a digital system using a high-level language, such as C or C++. The logic synthesis tool is similar to a compiler for hardware and maps HDL code onto a library of gates called standard cells to minimize area while meeting some timing constraints. This generates a Register Transfer Level (RTL) description of the system, which defines how data flows between registers and how computations are performed using Arithmetic Logic Units (ALU) or even just logic gates that implement Boolean functions or arithmetic. The RTL description is then synthesized onto gates and flip-flops to generate a Gate-Level Synthesis (GLS) netlist. The netlist is the lowest level of abstraction and represents the actual hardware implementation of the digital system.

As computation block nodes group these essential logic operations, the logic gate layout can be derived from the computation block nodes, as each is composed of basic gates like NOT, AND, OR, XOR, NAND, NOR, and XNOR, or more complex gates like multiplexers, decoders, adders, subtractors, shifters, multipliers, and dividers. From the fact that computation block nodes are already a series of elementary operations, they can also be seen as a sequence of Boolean functions and/or arithmetic.

In a first step, described more detailed below, the present inventive system transforms a source code into a sequence or code 32 of elementary operations

321,…,325 structured in loops, branches, and sequences. It is independent from platform and compiler optimization level, and hence the same transformation can be used to optimize execution time on any platform.

The present approach is based on decomposing a piece of source code 31 written in a programming language to elementary operations 32/321,…,325, i.e. distinct transformed parts of the source code 31. The set of elementary operations is finite for each processing unit with several subsets: integer, floating point, logic, and memory operations. These sets are corelated to parts of the architecture of a processor and the memory data-paths. The elementary operations, as used herein, can e.g. be classified in various levels as follows: The top level contains four operation classes: INTEGER, FLOATING POINT, LOGIC, and MEMORY. Second level of classification can be based on the origin of operands (i.e. location in memory space): local, global or procedure parameters. Each group can show different timing behavior: local variables, being heavily used, are almost always in cache, while global and parameter operands must be loaded from an arbitrary address and can cause a cache miss. Third level of classification is by operand type: (1) scalar variables and (2) arrays of one or more dimensions. Pointers are treated as scalar variables when the value of pointer is given using a single variable, or as arrays when the value of pointer is given using multiple variables. Operations which belong to INTEGER and FLOATING POINT classes are: addition (ADD), multiplication (MUL) and division (DIV). LOGIC class contains logic operations (LOG): (i.e. AND, OR, XOR and NOT) and shift operations (SHIFT): operations that perform bitwise movement (e.g. rotation, shift, etc.). Operations in MEMORY class are: single memory assign(ASSIGN), block transaction (BLOCK) and procedure call (PROC). MEMORY BLOCK represents a transaction of a block of size 1000 and it can only have array operands. MEMORY PROC represents a function call with one argument and a return value. Arguments can be variables and arrays, declared locally or given as parameters of the caller function, but not global.

(viii) "Processor" – "Cores"

To define the term "*core*", as e.g. a "multi-core processor", as used herein, a definition of the term "*processor*" (or "*microprocessor*") has to be given as well. In computing terms, a processor is a component that reads and executes program

instructions. Processor cores are processors, i.e. individual processing units, within a computer's central processing unit (CPU). The instructions for processors are technical signals to a processor/core 2102/2103 telling the processor/core 2102/2103 what to do, such as reading data from memory or sending data to an output bus. A common type of processor is the Central Processing Unit (CPU). A multi-core processor is generally defined as an integrated circuit to which two or more independent processors (called cores) are attached. It is to be noted, that this term is distinct from but related to the term multi-CPU, which refers to having multiple CPUs which are not attached to the same integrated circuit. In the prior art, the term uniprocessor generally refers to having one processor per system [Uniprocessor], and that the processor has one core. The term is used to contrast with multiprocessing architectures, i.e. either multi-core, multi-CPU, or both. Multi-core processors emerged in the computing industry from uniprocessor technology as a method to achieve greater performance through parallelism rather than raw clock speed. For years, the computer industry developed faster and faster uniprocessors, though this pursuit is drawing to a close due to the limits of transistor scaling, power requirements, and heat dissipation. Because single-threaded cores are reaching a plateau of clock frequency, chip manufacturers have turned to multi-core processors to enhance performance using parallelism.

(ix) Parallelization types and performance in multi-core CPU

Because multi-core CPUs fundamentally rely on parallelism to enhance performance, an understanding of the key types of parallelism is important to analyzing performance and understand the technical problems associated with auto-parallelization systems. Modern multi-core processors without code parallelization does not make sense. However, parallelism is a technically extremely challenging and complex object, however, in the context of the present invention, an understanding of three basic types of parallelism is sufficient here. Instruction-level parallelism, thread-level parallelism, and data-level parallelism are all employed by various multi-core CPU architectures, and have different impacts on performance that must be understood to conduct thorough performance analysis.

Instruction-level parallelism (or ILP) is a first type of parallelism, which involves executing certain instructions of a program simultaneously which would

otherwise be executed sequentially, which may positively impact performance depending on the instruction mix in the application. In the prior art, many CPUs utilize instruction-level parallelization techniques such as pipelining, superscalar execution, prediction, out-of-order execution, dynamic branch prediction or address speculation. However, only certain portions of a given program's instruction set may be suitable for instruction-level parallelization, as the example illustrates below in the following table 2.

Sequential Execution	Instruction-Level Parallelization
1. a = 10 + 5 2. b = 12 + 7 3. c = A + B	1. a = 10 + 5 1. b = 12 + 7 2. c = A + B
Instructions: 3 Cycles: 3	Instructions: 3 Cycles: 2(-33%)

Because steps 1 and 2 of the sequential operation are independent of each other, a processor employing instruction-level parallelism can run instructions 1.A. and 1.B. simultaneously and thereby reduce the operation cycles to complete the operation by 33%. The last step must be executed sequentially in either case, however, as it is dependent on the two prior steps. It is clear, that this example is an oversimplification. However, for the auto-parallelization system, the key is to extract which portions of an application have instructions that could run in parallel.

Thread-Level (respectively Task-Level) Parallelization (TLP) is a second type of parallelism involving executing individual task threads delegated to the CPU simultaneously. Thread-level parallelism can substantially impact multi-threaded application performance through various factors, ranging from hardware-specific, thread-implementation specific, to application-specific, and consequently a basic understanding is important for this type of parallelization. Each thread maintains its own memory stack and instructions, such that it may be thought of as an independent task, even if in reality the thread might not really be independent in the program or operating system. Thread-level parallelism is used by programs and operating systems that have a multi-threaded design. Conceptually, it is straightforward to see why thread-level parallelism can increase performance. If the threads are truly independent, then spreading out a set of threads among available cores on a processor reduces the elapsed execution time to the maximum execution time of any of the threads, compared to a single threaded version which would require additive execution time of all of the threads. Ideally, the work would also be evenly divided

among threads, and the overhead of allocating and scheduling threads is minimal. In the real-world, this simplistic ideal model of thread-level parallelism performance is complicated by several technical factors, such that the ideal scenario is rarely observed in real applications. Performance-impacting factors include the load balance, level of execution independence, thread-locking mechanisms, scheduling methods, and thread memory required. Further, data-level parallelism among the distributed threads may impact performance. The thread implementation library in both the operating system and the specific application will also impact performance.

Data-Level Parallelization (DLP) is a third type of parallelism involving sharing common data among executing processes through memory coherence, improving performance by reducing the time required to load and access memory. Typically, identifying application areas utilizing data-level parallelism will assist in understanding performance characteristics on multi-core processors.

Loop-Level Parallelization (LLP) is a fourth type of parallelism where if iterations of loops have no dependencies, they are executed in parallel. If all loop iterations require the same execution time, the distribution can be performed by a static distribution with a fixed number of iterations to each computing unit.

In the context of a multi-core CPU, data-level parallelism in the cache memory shared by cores can have a substantial impact on performance. Here, the executing processes running on multiple cores will be called threads. Performance gains are expected when the threads read from the same data in the shared memory. This scenario allows one copy of the data to be used by the multiple threads, reducing the number of copy operations and thus execution time. When the threads have no data in common, each thread must maintain a copy of its data and no gains are available. However, if the multiple requests to this memory exceed its bandwidth, increasing threads may produce negative performance impacts. Further performance impacts may also occur during write operations. Multiple threads attempting to write to the same memory location at the same time must wait to resolve conflicts. To cope with this problem, schemes to handle such situations are needed, such as spin-locks. The performance impact depends upon the penalties involved with the scheme employed and how often such conflicts occur. Generally speaking, having threads write to different areas of shared memory would be preferable in lessening the likelihood of incurring these penalties. Non-Uniform Memory Architecture (NUMA) can

help, as it places data used by one particular core physically closer to that core in memory. Further, bandwidth is here also an important factor as the number of threads increases on a multi-core processor. Limited cache size (cache misses), limited bandwidth, off-cache latency, and other aspects will have impacts on performance, though data-level parallelism can improve performance in certain situations. Further, the interaction between instruction-level and data-level parallelism affects performance. Flynn's taxonomy provides a technical framework to analyze these interactions. In summary, observations about data-level parallelism in a particular application can be important to analyzing its performance on multi-core CPUs because memory is frequently the limiting factor.

(x) P and NP complete auto-parallelization measure

In the field of computer technology, code parallelization, and processor architecture, a large numbers of systems and method deal with the technical problem of improving speed and memory space taken to compute codes and algorithms, as faster algorithms give extra time for more computations, and less space taken may further enhance computing, in particular parallel computing. As mentioned above, the time it takes a computer to compute a problem is called its "*time complexity*", while the space taken is called its "*space complexity*". The time and space taken to perform these algorithms are usually measured in the size of the input and the number of elements the algorithm has to manipulate. There is another measure linked to the two mentioned. The so called "*computational complexity*" is a measure for classifying computational problems according to their resource usage, and relating these classes to each other. A computational problem is a task solved by a computer and its processors, respectively. A computation problem is a technical problem solvable by mechanical application of processing steps by data processing means as processors and data repositories.

A computational problem is regarded as inherently difficult if its solution requires significant resources, whatever the processing steps or algorithm used. To provide the measure, inter alia, models of computation are used to quantifying the computational complexity, i.e., the amount of resources needed to solve a certain problem, such as time and storage. Other measures of complexity are also used, such

as the amount of communication (used in "*communication complexity*"), the number of gates in a circuit (used in "*circuit complexity*") and the number of processors (used in parallel computing).

One of the roles of the computational complexity is to provide a measure for the practical limits of a computers or a parallelization. On the other side, if a computational problem is regarded as easy to be solved, it means that the problem (here auto-parallelization) can be quickly solved using the computer means. The term *quickly*, used above, means the existence of a processing code or algorithm solving the task runs in polynomial time, such that the time to complete the task varies as a polynomial function on the size of the input to the algorithm (as opposed to exponential time). The general class of questions for which some algorithm can provide a solution in polynomial time is called "P", for polynomial. Thus a parallelization problem where the time to find a solution (i.e. a parallelized code) increase in time and space polynomial to the input, n, such as e.g. n^1, n^2 or n^{99} etc., where n is being raised to a power, belongs to the class of "P". On the other hand, for the class of parallelization problems, for which the time to provide a solution increases exponentially, i.e. not in polynomial time is called "NP" (Nondeterministic Polynomial time). NP parallelization problems, unlike parallelization problems in P, are parallelization problems that take computers an extremely long time and space to solve; time that grows exponentially as the number of elements in the input increases. This exponential time is described as any number is raised to the n^{th} power, such as 2^N.

In this context, it is to be noted that for some problems, there is no known way to find a solution quickly, but if one is provided with information showing what the answer is, it is possible to verify the answer quickly. In terms of the measure of the *computational complexity*, such a technical problem is called NP-complete ("Nondeterministic Polynomial-time complete") where "nondeterministic" refers to nondeterministic Turing machines, "complete" refers to the property of being able to simulate everything in the same complexity class, and "polynomial time" refers to an amount of time that is considered "quick" for a deterministic method to check a single solution, or for a nondeterministic Turing machine to perform the whole search, respectively. That means, a computational problem is NP-complete, if (i) it is a technical problem for which the correctness of each solution can be verified quickly (namely, in polynomial time) and a solution can be found by using a brute-force search by trying all possible solutions; and (ii) the problem can be used to simulate every other problem

for which it is possible to verify quickly that a solution is correct. In this sense, NP-complete problems are the hardest of the problems to which solutions can be verified quickly. If a system or algorithm is able to find solutions of some NP-complete problem quickly, this system or algorithm is able to quickly find the solutions of every other problem to which a given solution can be easily verified. In the computational technology, this problem here is called "*P versus NP problem*" which is whether problems that can be verified in polynomial time can also be solved in polynomial time. The problem has, inter alia, profound technical implications for parallelization of code and powerfulness of a system or method used for the parallelization. In particular, if a parallelization system and/or method is able to provide parallelization of a certain code or code structure, where this code or code structure is a NP-complete problem, all other similar NP-complete codes and code structures can be parallelized with this parallelization system or method to the same extent, with the exception of the polynomial reduction time. It is clear, that for parallelization of a code, in principle always brute force can be used for the search of the best optimized parallelization by generating all possible permutations of all possible basic blocs or computational block nodes or tasks or threads comprising all possible permutation of instructions in parallel and test it. However, using brute force, the complexity of code parallelization often becomes exponential having the result of being not any more scaling to the computational capacity of applied computer means. Some prior art solutions try to do branch and bound with different tasks or threads taking different choices or assumptions. But coordinating the tasks/threads becomes more difficult, and it becomes technically challenging to derive and/or verify the best optimized parallel code.

Detailed Description of the Preferred Embodiments

Figures 12 and 13 schematically illustrate an architecture for a possible implementation of an embodiment of the compiler system 1 for optimized compiling of a program code 3 for execution by a parallel processing system 2 having a plurality of processing units 21. It has to be noted that, herein, when an element is referred to as being "connected to" or "coupled to" another element, it can be directly connected or coupled to the other element or intervening elements may be present. In contrast, when an element is referred to as being "directly connected to" or "directly coupled to"

another element, there are no intervening elements present. Further, some portions of the detailed descriptions which follow are presented in terms of procedures, logic blocks, processing, and other symbolic representations of operations on data bits within a computer memory. These descriptions and representations are the means used by those skilled in the data processing arts to most effectively convey the substance of their work to others skilled in the art. In the present application, a procedure, method, logic block, process, or the like, is conceived to be a self-consistent sequence of steps or instructions leading to a desired result. Embodiment variants described herein may be discussed in the general context of processor-executable instructions residing on some form of non-transitory processor-readable medium, such as program codes or code blocks, executed by one or more processors or other devices. Generally, program codes include routines, programs, objects, components, data structures, etc., that perform particular tasks or implement particular abstract data types. The functionality of the program codes may be combined or distributed as desired in various embodiments. The techniques described herein may be implemented in hardware, software, firmware, or any combination thereof, unless specifically described as being implemented in a specific manner. Any features described as modules or components may also be implemented together in an integrated logic device or separately as discrete but interoperable logic devices. If implemented in software, the techniques may be realized at least in part by a non-transitory processor-readable storage medium comprising instructions that, when executed, performs one or more of the methods described above. The non-transitory processor-readable data storage medium may form part of a computer program product. For a firmware or software implementation, the methodologies may be implemented with modules (e.g., procedures, functions, etc.) having instructions that perform functions described herein. Any machine readable medium tangibly embodying instructions may be used in implementing methodologies described herein. For example, software codes can be stored in a memory and executed by one or more processors. Memory can be implemented within the processors, as e.g. registers, or external to the processors.

The various exemplary logical blocks, modules, circuits and instructions described in connection with the embodiments disclosed herein may be executed by one or more processors or processor units 21, such as one or more Central Processing Units (CPU) 210, e.g. comprising a control unit 2101, a processor 2102 with registers 21021 and combinational logic 21022, and/or Graphics Processing Units (GPU) 211, and/or Sound chips 212 and/or Vision Processing Unit (VPU) 213, and/or Tensor

Processing Units (TPU) 214 and/or Neural Processing Units (NPU) 215, and/or Physics Processing Units (PPU) 216, and/or Digital Signal Processors (DSP) 217, and/or Synergistic Processing Units (SPU) 218 and/or Field-Programmable Gate Arrays (FPGA) 219 or any other processor units 21 known in the art as e.g. motion processing units (MPUs) and/or general purpose microprocessors and/or Application Specific Integrated Circuits (ASIC) and/or Application Specific Instruction Set Processors (ASIP), or other equivalent integrated or discrete logic circuitry. The term "processor" or "processor unit" 21 as used herein may refer to any of the foregoing structure or any other structure suitable for implementation of the techniques described herein. In addition, in some aspects, the functionality described herein may be provided within dedicated software modules or hardware modules configured as described herein. Also, the techniques could be fully implemented in one or more circuits or logic elements. A general purpose processor 21 may be a microprocessor, but in the alternative, the processor may be any conventional processor, controller, microcontroller, or state machine. In the described embodiments, processing elements refers to a plurality of processors 21 and the associated resources such as memory or memory units 22. Some example methods and apparatuses disclosed herein that may be implemented, in whole or in part, to facilitate or support one or more operations or techniques for processing code in a plurality of processors. The multi-processing system 2 can also comprise processor arrays comprising a plurality of processors 21. Each processor 21 of the processor array may be implemented in hardware or a combination of hardware and software. Processor arrays can be representative of one or more circuits capable of performing at least a portion of information computing technique or process. By way of example but not limitation, each processor of a processing array can include one or more processors, controllers, microprocessors, microcontrollers, application specific integrated circuits, digital signal processors, programmable logic devices, field programmable gate arrays, or the like, or any combination thereof. As mentioned, the processors 21 can be any of general purpose Central Processing Unit (CPU), or special purpose processors such graphics processing unit (GPU)s, Digital Signal Processors (DSP), video processors or any other special purpose processors.

The invention comprises a compiler system 1 with sub-systems 11,…,16. In a non-limiting embodiment, the sub-systems comprise at least lexer/parser 11 and/or an analyzer 12 and/or a scheduler 13 and/or a matrix module 14 and/or an optimizer module and/or a code generator 16. Further, they can also comprise processor arrays and/or memory. Compiler 1 segments code into code blocks. For the described

embodiments, block or code blocks refer to a section or portions of code grouped together. Grouping enables groups of statements/instructions to be treated as if they were one statement, and to restrict the scope of variables, procedures and functions declared in a block so that they do not conflict with variables having the same name used elsewhere in a program for different purposes.

The above-mentioned memory or memory units 22 of the parallel processing system 2 can comprise any memory to store code blocks and data. Memory 22 may represent any suitable or desired information storage medium. Memory may be coupled with processor units 21 and/or processing array. As used herein the term "memory" 2 refers to any type of long term, short term, volatile, nonvolatile, or other memory and is not to be limited to any particular type of memory or number of memories, or type of media upon which memory is stored. Memory 2 can e.g. comprise primary storage units 211 as processor registers 2111 and/or processor caches 2112 comprising multi-level caches as L1-cache 21221, L2-cache 21222, etc. and/or Random Access Memory (RAM) units 2113. In respect to this application, it has to be noted that the issue of the multi-level caches lies in the tradeoff between cache latency and hit rate. Larger caches have better hit rates but longer latency. To address this tradeoff, multiple levels of cache can be used, with small fast caches backed up by larger, slower caches. Multi-level caches generally operate by checking the fastest, level 1 (L1) cache first. If it hits, the processor can proceed at the higher speed. If that smaller cache misses, the next fastest cache (level 2, L2) is checked, and so on, before accessing external memory. Cache access cannot be controlled directly by a programmer. It can be influenced by data locality and / or by compiler hints. As the latency difference between main memory and the fastest cache (cf. figure 1) has become larger, some processors have begun to utilize as many as three levels of on-chip cache. The memory 2 can further e.g. comprise secondary storage units 212 and/or third storage units 213 (e.g. tape backup etc.), the secondary storage units 212 comprising e.g. Hard Disk Drives (HDD) 2121 and/or Solid-State Drives (SSD) 2122 and/or Universal Serial Bus (USB) memory 2123 and/or flash drives 2124 and/or optical storage devices (CD or DVD drives) 2125 and/or floppy disk drives (FDD) 2126 and/or RAM disks 2127 and/or magnetic tapes 2128 etc. In at least some implementations, one or more portions of the herein described storage media may store signals representative of information as expressed by a particular state of the storage media. For example, an electronic signal representative of information may be "stored" in a portion of the storage media (e.g., memory, register, flip-flop, etc.) by affecting or changing the state

of such portions of the storage media to represent information. As such, in a particular implementation, such a change of state of the portion of the storage media to store a signal representative of information constitutes a transformation of storage media to a different state or thing. As mentioned above, memory 2 can e.g. comprises random access memory (RAM) such as synchronous dynamic random access memory (SDRAM), First-In-First-Out (FIFO) memory, or other known storage media.

(Micro-) processors are based on integrated circuits, which makes it possible to do arithmetic and logical operations based on (two) binary values (in the simplest 1 / 0). For this, the binary values must be available for the processor's calculating unit. A processor unit needs to get the two binary value to calculate the result of an expression a = b operand c. The time it takes to retrieve the data for these operations is known as the latency time. There is a wide hierarchical range for these latency times from the register, L1-cache, memory access, I/O Operation, or a network transfer, as well as processor configuration (e.g. CPU vs. GPU). As every single component has a latency time, the overall latency time for the calculation is a combination mainly of hardware components needed to get the data from one location to the other in a modern computation infrastructure. The difference between fastest location for a CPU (or GPU) to get data and the slowest can be huge (in the range of factor >10^9).

Latency from a general point of view is the time delay between the cause and the effect of some physical change in a system being observed or measured. The latency, as used herein, is directly related to the physical structure of the multi-processing system 2. Multi-processing systems 2 comprise processor units 21 based on integrated circuits, which makes it possible to do arithmetic and logical operations based on (two) binary values (in the simplest 1 / 0). Those binary values must be available for the processor's calculating unit. A processor unit needs to get the two binary value to calculate the result of an expression a = b operand c. The time it takes to retrieve the data for these operations is known as the latency time. There is a wide hierarchical range for these latency times from the register, L1-cache, memory access, I/O Operation, or a network transfer, as well as processor configuration (e.g. CPU vs. GPU). As every single component has a latency time, the overall latency time for the calculation is a combination mainly of hardware components needed to get the data from one location to the other in the multiprocessing system's infrastructure. It is worth noting that the speed of microprocessors has increased by more than a factor of ten per decade, but the speed of commodity memories (DRAMs) has only doubled, i.e.,

access time is halved. Therefore, the latency of memory access in terms of processor clock cycles grow by a factor of six in 10 years. Multiprocessor systems 2 intensify the problem. In bus-based systems, the establishment of a high-bandwidth bus between the processor and the memory tends to increase the latency of obtaining the data from the memory. When the memory is physically distributed, the latency of the network and the network interface is added to that of the accessing the local memory on the node. Latency usually grows with the size of the multiprocessing machine 1, as more nodes imply more communication relative to computation, more jump in the network for general communication, and likely more contention. The main goal of parallel computing hardware design is to reduce the overall used latency of the data access by maintaining high, scalable bandwidth while the main goal of parallel processing coding design is to reduce the overall idle time of the processor units 21. In general, idle time of the processor units 21 may have several causes, as the latency of memory access, deadlocks, or race conditions, e.g. if the sequence or timing of the code blocks or threads processed by the processor units 21 depend mutually, i.e. depend on the relative timing between interfering threads. As used herein, deadlock is a state in which a member of the processor units 21 is waiting for another member's output, e.g. the output of an instruction block processed by another processor unit 21, to take action. Deadlock is a common problem in multiprocessing systems 1, parallel computing, and distributed systems, where software and hardware locks are used to arbitrate shared resources and implement process synchronization. Thus, a deadlock, as used herein, occurs when a process or thread enters a waiting state because a requested system or data resource is held by another waiting process or not yet achieved by the process, which in turn can be waiting for another resource or data held by another waiting process. If a processor unit 21 is unable to further process because the resources requested by it are being used by another waiting process (data access or output of a not yet finished process of another processor unit 21), then this is denoted herein as a deadlock leading to idle time of the respective processor unit 21.

The compiler system 1 comprises means for translating a source programming language 31 of the computer program 3 into a machine code 32 as target programming language generating a processing code 3.1, ... , 3.n comprising a number of instructions executable by the plurality of processing units 21 of the parallel processing system 2 or controlling the operation of the plurality of processing units 21. The source programming language can e.g. be a high-level programming language

31. The high-level programming language 31 can e.g. comprise C and/or C++ 311 and/or python 312 and/or Java 313, Fortran 314, OpenCL (Open Computing Language) 315 or any other high-level programming language 31. It is important to note, that the auto paralleling compiler system 1 can also be applied to machine code 31 or assembler code 31 as source code to achieve parallelization of the code. In this case, the translation of the high-level language to machine code instructions does not have to be performed by the compiler system 10.

The parallel processing system 2 comprise memory units 22 comprising at least main execution memory units 221/2212 comprising a plurality of memory banks for holding data of at least parts of the processing code 32, and transition buffer units 221/2211 comprising high speed memory for storing starting locations of the processing code 32 and data segments at least comprising branch or jump instructions and/or used memory references and data values, wherein the main execution memory units 2212 provide a slower access time than the transition buffer units 2211. The transition buffer units 2211 can e.g. comprise cache memory modules 2211 and/or L1-cache 22121.

The execution of the processing code 32 by the parallel processing system 2 comprises an occurrence of latency times 26, the latency times being given by idle time of a processing units 21 for retrieving and/or saving data necessary for execution of a specific block of instructions of the processing code 32 by the processing units 21. The latency time can e.g. comprise register 2211 access time and/or L1-cache 22121 access time and/or memory 2213 access time and/or I/O operation time and/or data network transfer time and/or processor configuration time.

The compiler system 1 comprises a parser module 11 for translating the source programming language 31 into a code 32 of elementary instructions directly executable by the processing units executable by the number of processing units 21, the elementary instructions selectable out of a processing unit specific set of elementary instructions comprising arithmetic operations 321 and/or logic operations 322 and/or control operations and/or I/O operations, in particular variable and array declarations instructions 323, compare operation instructions 324, and code flow instructions 325, for the number of processing units 21. The arithmetic operations 321 can e.g. comprise the operations addition, subtraction, multiplication, and division. The logic operations 322 can e.g. comprise a number of logical expressions like equal, not

equal, greater than, less than, greater than or equal, less than or equal. The control operations can e.g. comprise a "branch expression" and/or a "loop expression".

As an embodiment variant, at least two of the processing units can e.g. have different sets of elementary instructions. The different processing units having different sets of elementary instructions can e.g. comprise Central Processing Units (CPU) 210, Graphics Processing Units (GPU) 211, sound chips 212, Vision Processing Unit (VPU) 213, Tensor Processing Units (TPU) 214, Neural Processing Units (NPU) 215, Physics Processing Units (PPU) 216, Digital Signal Processors (DSP) 217, Synergistic Processing Units (SPU) 218, Field-Programmable Gate Arrays (FPGA) 219 etc.

The parser module 11 comprises means for partitioning the code of elementary instructions into computation block nodes 333 each consisting of a smallest possible segmentation of non-further decomposable units each comprising a sequence of the elementary instructions requiring the same input data. Two or more computation block nodes 333, each with chains of elementary instructions, form a computation chain 34 creating an ordered flow of operations/instructions on the input data. The chains in the computation block nodes 333 (sequence of the elementary instructions) are built by a fixed rule: The instruction is placed at the position in the chain 34 where the new elementary instruction READs after the elementary instruction that WRITES the data point. This automatically forms a chain 34, which is data centered and maps the necessary, physically limited READ and WRITE operations in a computational register 2211, L1-cache 2212, network I/O or similar of a CPU 21.

The compiler system 1 comprises a matrix builder 15 for generating a number of numerical matrices 151,...,15i out of the computation chains 34 depending on the latency times 26. In the graph from the computation chains 34, the dependencies of the system can be assessed, however they cannot simply be broken down into individual independent chains. There exist fusion and split situations of computation block 333 chains in the chains 34, resulting from data dependencies and/or code branches. The latency to distribute information in the hardware system is introduced as a physical time. By assigning each computation block node 333 at least this time interval length, each computation block node 333 in the graph or tree structure can be numbered according to its position in the graph and given a block number and therefore giving computation block nodes 333 with same "temporal position" in the graph the same number. If a computation block node 333 has to be at

least the length of the time it would take to distribute information in the system and should as many as instructions as possible have to be calculated during this time and each computation block node 333 has a number based on its position in the graph depending on the program flow, a set of matrix 151, ..., 15i can be build.

The matrix 151,…,15i also shows that they can be mathematically captured (a graph model based on CBs, CCs, etc. is not simply captured as a table/matrix). Thus, the data operations and communication are given and build numerical matrices, which can be optimized (changed) e.g. by using ML or AI, depending on the target platform and/or hardware setup.

The compiler system comprises a numerical matrix optimization module 16 using a numerical matrix optimization technique minimizing an overall occurring latency time as aggregated latency times 26 by providing an optimized structure of computational chains 34 being processed by the plurality of processing units 21, wherein by means of a code generator 17 an optimized machine code is generated for the plurality of processing units of the parallel processing system having an optimized overall latency times 26. The optimization can now be applied to the hardware infrastructure. The quantities that are important for the optimization are known in numerical form in the matrices 151,…,15i for each time unit and for each independent chain and branch: E.g. from the contents in the matrices: (i) number of elementary instructions, which must be sequential, (ii) size of the data transfer from the computational block x in computation chain u and when this transfer is needed again on computational block y in computation chain v (where y > than x) (e.g. possible via network or combine computation blocks to let data be on the same cacheline). As an example: for a GPU the data should be copied from memory to GPU memory in one process step and then all elementary instructions with the same properties should be executed at once, but on CPU, operations with the same data should be on the same cache-line (CPU dependent) or operations of specific data types can be calculated on the corresponding CPUs with the necessary better instruction sets. In the inventive system, this always leads to a parallel code even without optimization, because the elementary elements are grouped in sequential groups in the CBs.

In summary, since (micro-) processors understand only elementary instructions, the source code 31 is split into these elementary instructions in order to achieve the most basic level of parallelization. (It is to be noted, that the present

invention is also applicable to the technical problem of optimization of (micro-) processors being based on the principle of integrated circuits (IC), which are a set of electronic circuits. The mentioned instructions are linked with the configuration of the electronic circuits on the (micro-) processor and therefore the following topics are also applicable to any form of integrated circuits or vis-versa can be used to derive optimized integrated circuits (or configuration of electronic circuits or directly electronic circuits) for a given code, as a instruction can be seen as a form of a configuration of an electronic circuit representing a computation operation (e.g. +, -, manipulate data, etc.)). The following core points are key to the inventive system: (1) Elementary instructions of a processor system are combined according to their unique "READ" and "WRITE" behavior, i.e. forming a chain of instructions according to the rule: "*an instruction writes to X1, a new instruction which reads to X1 is appended after the last instruction which writes to X1*", in a chain of nodes and links to computation blocks 333; (2) If it is necessary to propagate an information/data in the multiprocessor system 2, a new *computation block* 333 starts; (3) Each computation block 333 has a minimum time length. This is proportional to the length of time (latency), which is needed to propagate information (data or signal) to/from the *block* 333 in the hardware system; (4) In case of a graph model with two read data nodes, instruction node, and one write data node, which are connected by links, chains from the computation blocks 333 have places where two chains: (a) meet (e.g. because an instruction reads from two data points, which are written in two different *computation blocks* 333 or because a *branch* exists), (b)occur, e.g. if two computation blocks 333 can be started by reading simultaneously. If necessary, a graph model could e.g. also be based on more than 2 read nodes and several write nodes or combining several instructions in one operation/instruction node; and (5) These chains 34 can be broken down and the instructions and the necessary information transfers per discrete time interval could be captured in matrices. For example, each row is divided into columns (time interval one column) and contains the independent chains of instructions and the necessary information transfers to others. These are thus tangible for auto-optimization processes, in particular numerical optimization processes. This provides the basis for a fully automatic parallelization of source code. It has to be noted, that in the present auto-parallelization system 1, the graph model is not just based on a matrix or table representation but provides, inter alia, a multi-dimensional nested tree structure of a computation block graph, as task graph, associated to computation parallel chains 34, allowing the system 1 to assess properties that can be exploited for auto-parallelization, code optimization, computation block 333 scheduling, and even automated cost

estimation or automatic mapping to different architectures of multi-processing system 2.

As mentioned, in the computation matrix, the cells containing the chain 34 of instructions given by the sequence of computation block nodes 333 forming the chain 34, and in the transfer matrix, the cells contain the transfer properties, i.e. the needed transfers to and from other computation block node 333. It is to be noted, that the transfer properties comprise of what information is needed in another computation block node 333. Depending on the level of the target infrastructure, this can be solved by classical compilation/controlled by the processor = transfer in cache (classic compiler distribute the data to register and try to exploit data locality to use cache-levels efficiently) or explicitly shared by shared memory and guarded by a lock or explicitly send and received from/to different nodes in a cluster using e.g. the Message Passing Interface (MPI) protocol or send and received by a socket in a multicore Inter-Process Communication (IPC), etc.. The transfer properties may have any form of communication, ranging from handled by the processor (cache) or explicit by a communication pattern (e.g. IPC via queue, MPI, etc.). This makes the inventive method scalable to a wide range of platforms and/or infrastructures. Thus, the transfer properties, for example, may comprise information as send data %1 (integer) to cell (1,7). Transfers and/or communications need a certain time, which typically directly dependent on the (transfer/communication) latency times in a particular system. Finally, it has to be noted, that, as mentioned, in a specific row of the computation matrix, the column cells comprise the flow or sequence of instructions wherein each column cell of the row comprises one computation block node 333 of the sequence of computation block nodes 333, forming the chain 34 of the specific row. However, in specific embodiment variants, not each cell of a row necessarily needs to comprise a sequence of instructions or an instruction. One or more of the cells of one or more specific rows of a computation matrix can also be empty. This is also true for the transfer matrix. Computation matrix and transfer matrix are normally equal in their size, i.e. number of rows and columns. The computation matrix and the matrix and transfer matrix providing a possible technical structure for the auto-parallelization.

Technically, the matrix builder connects the computation block nodes 333 according to the program flow (e.g. fig. 35) and/or the code 32, respectively, from the parser module 11. The program flow is represented by connecting the computation block nodes 333 (similar to basic blocks), e.g. when adding a new instruction

(=operation node), a new computation block node 333 can e.g. be generated, where, in general, there is no clear dependency (cf. figure 21), the new computation block node 333 being positioned after another computation block node 333 or under a branch node. After adding all instructions, it is possible to numbering each computation block node 333 along the program flow forming the sequence of the so generated chain 34. These numbers (block numbers) are equal to the column of the matrix. Computation block nodes 333 with same block numbers are distributed to different rows in the same column of the computation matrix.

Generation of tasks and gamma structure/graph

Up to here, it is explained how to group basic instructions from sequential source code (a list of serial instructions) into the herein defined computation block (CBs = list of instructions preferred to run in serial) and indexing the resulting groups to retrieve groups of instruction to run in parallel including the needed transfers to run on different computing units. The method is built on a systemic perspective of modern computing platforms, which still are limited by binary computing steps. This basic concept, how a mathematical operation, such as +, -, *, /, etc. is computed on a binary computing system, i.e. a digital processor 2102/2103, results in some physical constraints and dependencies: (i) To compute a statement of a = b <op> c, the bit-pattern b and c each representing a number of corresponding accuracy (int to double precision) must be physical available to a binary computing unit in the moment of the calculation; (ii) To transfer a bit-pattern a, b or c on a platform or a system, a latency time $\Delta t_{transfer}$ arises; (iii) This $\Delta t_{transfer}$ is at least a power higher than computing several instructions in serial on one computing unit, i.e. on one processor 2102 or processor core 2103. The grouping bases on data-dependency extractions, resulting in grouping of instruction in computation blocks (CB). Each of them consists of one chain of sequential instructions. All needed data transfers between the CBs are kept. This composition makes it possible to optimize and auto-parallelize source code to be processed on parallel computing units, i.e. processors 2102 and processor cores 2103, respectively without any programmer's hint in a hardware-agnostic manner.

An embodiment variant uses the system and method is in the middle end of an auto-parallelization compiler, see figure 84. The generated result can be given out

as a tensor or a graph, depending on the form of representation, see figure 85. If the code is represented by a tensor (or matrices), one consists of the computation block with their chain of instructions in each cell. Each row represents a series of instruction groups to be computable independently from the grouped instructions in the other rows of the same column. In a second matrix the corresponding data transfers are gathered between the instruction groups. Figure 80 also illustrates how tensor entries can be represented as a graph. It has to be noted, that as a CB A is positioned in a row and column to the segment number earliest possible after the read data is available in CB B with a segment-number A=B+1 (1 CB correspond to a potential transfer time), continuous entries in computation matrix belonging to the same condition, are based on the same read-data. These continuous segments can be called compute segment and indicate the continuous CBs can be computed without the need of other information. In case between two gamma-nodes 1 edge exist, the gamma-nodes can be combined to one gamma-node (Γ_{3a}+ Γ_{3b} => Γ_{3b}). This makes it possible to combine them in the same gamma-node, as they base on the same data in the same way as continuous CBs on the same row if belonging to the same branch node. Such compute segments are by definition a computation block node 333, as they enclose an arbitrary series of sequential instructions not sharing any data dependencies. Applied to the Fibonacci case, this is visible in figure 85, and leads to the computation graph as in "*Fundamentals of parallel programming module parallelism*" V. Sarkar. The indication of the compare instruction in the example for the "unrolled case" is for illustrative purposes to illustrate how the method applied to LLVM-IR in figure 75 forms to a gamma-graph. In figure 30 is shown how the method form at least one continuous series of CB over different branch nodes, keeping the program-flow from calling a program (indicated with main()) to the end of a program (indicated with return(0)). This is also visible in figure 85. Interpreting this gamma-graph in a multithreaded platform, each edge means to introduce a latency time for a context switch.

(a) Combining rows/CBs to Tasks

As by the indexed version, the CBs which can be computed in parallel are in the same column, which means they have the same segment number. It is possible to combine CBs with same segment numbers (see figure 39). In this step the instructions in the new computation segment add up and possible transfers between the units

vanish, see Figure 11. This has an influence on the time to compute a CB in parallel or in serial.

Another representation is in form of a computation or task graph, which is called herein "gamma-graph". To distinguish:

(i) A ***Computation Graph*** is used in reference to (cf. Fundamentals of parallel programming module 1 parallelism, V. Sarkar; see above) where nodes consist of arbitrary sequential instructions in the field of ideal parallelism;

(ii) A ***task graph*** is a generic graph with tasks, independent how these tasks are formed (e.g., as a computation graph or in the sense of function in the field of functional programming or otherwise);

(iii) A ***gamma-graph*** is a graph where each node is one CB or a combination of CBs. A gamma-graph is a task graph and in initial-granularity a computation graph. A gamma-node represents a compute segment, see above.

Each node represents one or more CBs. If in a row of the tensor representation (meaning on one unit) if there are no entries for transfers between columns, the computation blocks can be combined to one compute segment. For each row a distinct compute and communicate sequence evolve. A gamma-node consists at least of one computation block node (CB) or a composition of several with the same segment number and or with the same compute segment. From this, a physically based, unique dynamic task-granularity of the code can be gained by the inventive system and method, as described below.

Further, as consequence of the decomposition with the method is getting in each row (see tensor representation) or nodes (see in the gamma-graph) distinguished segments to compute and communicate. From the gamma-nodes corresponding parallel code can be generated, see Figure 87. $\Delta \boldsymbol{t}_{\boldsymbol{compute}}$ is the computing time and $\Delta \boldsymbol{t}_{\boldsymbol{transfer}}$ the communication/transfer time. Figure 87 shows distinguished computing and transfer / communication sections. Both the computing time $\Delta \boldsymbol{t}_{\boldsymbol{compute}}$ nor communication/transfer time $\Delta \boldsymbol{t}_{\boldsymbol{transfer}}$ are easily to get nor must be fixed for a platform,

but it is a unique feature of the method to be able to extract these two distinguished properties from code during compilation. This makes it possible to optimize the distribution of the nodes (code-chunks) to available computing units by using available frameworks, e.g. openMP, MPI or others.

In summary, the inventive auto-parallelization method and system groups and indexes instructions in a sequential code and provides the code in form of grouped instructions computing and resulting communication (transfer of data) sections representable as a tensor or graph. The computation sections can be seen as tasks with corresponding transfers/communication to other tasks. The building of the tasks follows the fundamental physical based principle that distributing data (bit-information) on different computing units introduce transfer and can reduce the computing time. It is a method to generate physically based data chunks in different granularity from code. By the ability to combine only parallel CBs the method makes it possible to generate task graphs with different task granularity. The ability to generate for a given code such task graphs introduce new opportunities to automatically optimize/parallelize code to a given hardware consisting of more than one computing unit. Combined with a solution approach to solve the potential occurring NP-complete problem of job/task scheduling on a parallel machine, the inventive method is a generic method to distribute code to homogeneous or heterogeneous platforms.

The granularity of each task can be expressed as G = $\Delta \boldsymbol{t}_{compute}/\ \Delta \boldsymbol{t}_{transfer}$. For example, see "*Evaluation of parallel programs by measurement of its granularity*", J. Kwiatkowski. Granularity is linked to the levels of parallelism. There are several levels of parallelisms (see above): (a) Instruction parallelism: Instruction level parallelism (ILP) results in very fine-grained parallelism and is exploited by analyzing the instruction dependencies. The ILP is also implicitly exploited by the hardware scheduler of the microprocessor, i.e. the microcontroller; (b) Data parallelism: A lot of programs operations are applied to elements of larger data structures. The operations can be executed on parallel or distributed systems; (c) Loop-parallelism: If iterations of loops have no dependencies, they can be executed in parallel. If all loop iterations require the same execution time, the distribution can be easily performed by a static distribution with a fixed number of iterations to each computing unit; and (d) Task (or thread) Parallelization, which defines parallelization more in the sense of functional parallelism. Massively simplified, SOTA compilers are based on ILP and for some extend for loop-parallelism. It is one of the first steps in a SOTA compiler to decompose a code

into Basic Blocks. A basic block (BB) is a code sequence of instructions without interruption of jumps. Meaning there are no branches in except one for the entry and out at the exit (cf: Hennessy, John L.; David A. Patterson, *Computer architecture: a quantitative approach*, Elsevier, 2011). The inventive method extends the Basic Block concept and extracts for each BB possible parallel instructions and corresponding transfer options to and from other BBs in the corresponding Control Flow Graphs (CFG).

In the following, it is shown how the concept of Basic Blocks interact with the inventive method. Figure 88 shows schematically a code in form of BB and CFG. A corresponding graph representation of potential computation blocks (CBs) after applying the method to the code is illustrated in Figure 89. The grouping and sequence of the CBs is depending on the instruction data dependency within the BB. After indexing the CBs the computation and communication segments evolve for 2 units visible in Figure 90. T and F show the segments depending on the condition leading to a branch in Figure 89. The branching forms a transfer time $\Delta \boldsymbol{t}_{\boldsymbol{transfer}}$ visible in the first communication step in Figure 90, in case there are possible parallel instruction opportunities to parallelize the computation in the BB.

For each path (one possible order of nodes through code based on one specific combination of conditions) corresponding transfer sizes S_{path1}, S_{path2} and S_{data} are extracted, see figure 89. In a resulting parallel code, each branch transition, respectively change of Basic Block has corresponding transfers, depending on the instruction parallelism possibilities within each Basic Block. It is important to annotate, that by combining CBs, also these transfer properties may vanish, resulting in an optimization problem best optimizable by SOTA compiler methods targeting one computing unit. The computation segments can be seen as tasks and the transfers as communication. A code or program code is a sequence of operations with well-defined total order, where parallelism can be expressed as a partial order. A computation graph, as used herein, is what is called a Direct Acyclic Graph (DAG), where each node is a chain of sequential instructions (=task) and the edge are the ordering constraints. The inventive auto-parallelization method, inter alia, is able to extract a certain Computation Graph (CG) G from code (for Computation Graph see e.g. "*Fundamentals of parallel programming module parallelism*" V. Sarkar) automatically. Several properties are associated with a computation graph G, where $\boldsymbol{t}_{\text{runtime,p}}$ is the execution time in parallel and P the number of computing units. If it is

assumed that the execution time *time(N)* is known and it is an uninterrupted sequential computation, this does not depend on the scheduling on a parallel machine:

- Sum of execution times of nodes in G: $\mathrm{Work}(G) = \sum_{node\ N\ in\ G} time\ (N)$
- Longest path in G (critical paths) -> Critical path length: $CPL(G)$
- Most optimized (ideal) parallelism of a given computation graph G: $\mathrm{Work}(G)/CPL(G)$

For the scheduling, i.e. the action to distribute computational tasks to resources, the knowledge about length of each task and how they are linked to each other can be essential to fulfil their constraints and optimally schedule them. This can be particularly true for symmetric parallel architecture under the assumption that the computation time of a CB is small in respect to the transfer time. Here, the critical path (longest path in G = CPL) is an important property, providing a measure how well a given code can be parallelized, as it is not possible scheduling to compute or process the CG G in less than this time.

For the execution of G on a parallel processor machine with P processors (independent of the scheduling): (1) The capacity bound is at least $\boldsymbol{t}_{\text{runtime,p}} \geq \mathrm{Work}(G)/\mathrm{P}$. This means, a runtime less than $\boldsymbol{t}_{\text{runtime,p}}$ is not possible on P processors (even when divided perfectly); (2) The critical path bound is at least $\boldsymbol{t}_{\text{runtime,p}} \geq CPL(G)/\mathrm{P}$. This means, it is not possible for a certain code to schedule a parallelization in less than $\boldsymbol{t}_{\text{runtime,p}}$, as each scheduler must obey the dependencies in the critical path. Thus, a runtime in parallel $\boldsymbol{t}_{\text{runtime,p}}$ is given by:

$$t_{runtime,p} = max\left(WORK(G)/p, CLP(G)\right)$$

In summary, the present inventive system and method provides a new technical approach to extract a computation graph for a given code. With EV3 this is also possible for loop-sections without explicitly resolve every data-dependencies between the instructions. This enables new opportunities to auto-parallelization of source codes to a hardware depending on its latency characteristics. The present invention, in particular, shows is immense potential by being applied to the technical problem of loop-level parallelization, which is the central aspect in high performance

computing (HPC) applications as loop-sections can introduce large computing demands. Opportunities to exploit the inventive highly optimized parallelization is particularly given in application where data are stored in random access data structures. In this context, a well-known technical example to benchmark auto-parallelization is the recursive or loop implementation to generate Fibonacci numbers, which is discussed below. Further, the present auto-parallelization compiler is able to form task graphs as a function of a given transfer time (TT), when seeing the transfer time as the overhead for a parallel machine to run an application in parallel (e.g. context switch in a threaded environment). I.e. the present inventive system is able to provide a specifically auto-parallelization generating an optimized parallel code optimized on hardware specific characteristic to run tasks in parallel of the parallel-processing system used, in particular a specific architecture of one or more multi-core central processing units (CPU).

(b) Transfer-Matrix Elements and Computational Length of Tasks

In general, the method can be used to get from code a computational graph as defined in the field of ideal parallelism, compare "*Fundamentals of parallel programming module parallelism*" V. Sarkar. This enables to deduct some code characteristics when applying the assumptions the execution time is known for all nodes with uninterrupted sequential computations and this time is not depending on the scheduling and there are no limits in processors. As discussed above, for the optimal scheduling, i.e. the action to distribute computational tasks to resources, the knowledge about length of each task and how they are linked to each other can be essential to fulfil their constraints and optimally schedule them, i.e. achieve best optimization.

To measure the performance and hardware-specific parameters of a processor 2102 or processor core 2103 or parallel processor architecture measuring clock speed frequencies is not enough for the present inventive auto-parallelization system 1, but there are more accurate parameters needed to measure performance. It is important to understand that, nowadays, there are a significant number of processor types and brands in the market differing significantly in their hardware properties and characteristics: Intel, AMD (Advanced Micro Devices, Inc.), and ARM (Acorn RISC

Machines or Advanced RISC Machines) are examples. There are multiple different architecture types and tiers of processors for each company, which complicates things even more. For to best optimization of the parallelization, an accurate way is needed to compare these processors computing power. One possibility to measure processor performance is measuring Instructions per Second (IPS). Thus, IPS, today typically measured in MIPs (million instructions per second) or GIPs (billion instructions per second), can e.g. be used as a measure of a speed of a processor 2102/2103, and can be regarded as a general measurement or benchmark of how many instructions a processor 2102/2103 can handle in a single second. However, for example, for complex instruction set computers (CISCs), different instructions take different amounts of time, so the value measured depends on the instruction mix. Even for comparing processors in the same family the IPS measurement can be problematic. Many reported IPS values have represented "peak" execution rates on artificial instruction sequences with few branches and no cache contention, whereas realistic workloads typically lead to significantly lower IPS values. Memory hierarchy also greatly affects processor performance, an issue which is not proper considered by classic IPS measurements. Thus, there is no proper way of measuring MIPs, and MIPS measurements cannot be used as a measure of instruction execution speed as required by the present inventive system 1, but at the most as a task performance speed compared to a reference. In summary, the speed of a given processor depends on many factors, such as the type of instructions being executed, the execution order and the presence of branch instructions (problematic in processor pipelines) as well as different cache-levels.

It is to be noted again that processor instruction rates are different from clock frequencies, as each instruction may require several clock cycles to complete or the processor may be capable of executing multiple independent instructions simultaneously. MIPS can be useful when comparing performance between processors made with similar architecture (e.g. Microchip branded microcontrollers), but they are difficult to compare between differing CPU and processor 2102/2103 architectures. In particular, a MIPs measurement with a higher number does not mean much for the real-world situation of the operation of the present system 1 requiring exact hardware-specific measuring parameter values. Regarding the processor architecture, the design process involves choosing an instruction set and a certain execution paradigm (e.g. VLIW or RISC) and results in a certain microarchitecture, which is typically described in e.g. VHDL or Verilog. For microprocessor design, this description is then manufactured employing some of the various semiconductor device fabrication processes, resulting in

a die which is bonded onto a chip carrier. This chip carrier is then soldered onto, or inserted into a socket on, a printed circuit board (PCB). The mode of operation of any processor is the execution of lists of instructions. Instructions typically include those to compute or manipulate data values using registers, change or retrieve values in read/write memory, perform relational tests between data values and to control program flow. The already mentioned clock speed of a processor is another measure typically measured in megahertz and gigahertz. However, as also discussed above, clock speed by itself is also not an accurate way to gauge processor performance for the present invention. Finally, FLOPs (floating-point operations per second) are still another factor to measure the performance of a processor. A floating point number is a number that has floating decimal points, such as 0.008. However, a FLOPs benchmark only measures the floating point operations and not the integers, which means it too cannot solely gauge processor performance.

Measuring the execution time of single CBs alone is therefore not effective nor practicable. But sota compiler methods are built to optimize a series of instruction on a given processor, a NP-complete optimization. The presented inventive system does not change any indication on the scale how to schedule instructions on a processor. For the inventive system the relative computation time of CBs is much more relevant than the absolute compute time for one particular instruction or one CB. The goal is more to determine the relative duration of different CBs and split the computing time from data loading and storing (affected by memory hierarchies e.g., cache-accesses), which is expressed as transfers in method (e.g. in a multicore architecture data between computing units can be shared by L2 or L3 caches and the optimization of the inventive system is to distribute the CBs in the form the transfer occur over L2 / L3 caches). Sota compiler methods can be used to compile the code in a CB to machine code. Modern sota compiler methods are targeted to exploit processors characteristics, such as exploiting instruction parallelism by using e.g. appropriate specific registers (e.g. XMM registers), use floating-point units, out-of-order execution, etc. for a given series of instructions (mostly in the scope of a Basic Block). This results in optimized machine code for a target computing unit. As noted, established method exist to determine cycle duration of single machine instruction, which are a function of the processor clock speed, see A. Fog, "*Instruction tables*", Technical University of Denmark, 2022. Based on these tabled values for these machine instructions and the available latency times as a function of data size to data hierarchies (e.g. cache-level, memory, etc.), relative duration to compute a CB can be derived. The optimal use of

scheduling single instructions is covered in this way with the well-known sota compiler method, which are very well established and cover the NP-complete problem of scheduling instructions for one computing unit / processor. Such a step also derives the relation of a series of arbitrary sequential instructions and the needed data size S_{comp} to compute this set of instruction on a given hardware. In conjunction with clock speed frequencies, this allows to model compute time differences between distinct CBs (=series of sequential instructions). As CBs in the same column represent unique distinct set of instruction-chains and without any data dependency the relative runtime can be approximated by number of instructions, specific cycle number per instruction and access to memory. It has to be annotated, that instruction-chains in continuous CBs base always on the same unique first "read" information (within the CB, the read value for the next instruction is located directly after the write instruction = definition of a computation block node 333). Any other information needed to compute in a CB is gathered by transfers in the transfer matrix. This fundamental characteristic of data locality makes it possible to derive if this data could be on different memory hierarchies (cache, memory or disc or network, etc.). This differs for combination of CBs in the same column, as this indicates data dependencies exploitable, e.g. by cache-levels. This effect can be detected by cumulating data size of all parallel CBs, as this is the minimal size of data needed to be loaded to registers and the latency to corresponding cache-levels, which can hold the information, can be derived. As data is loaded in chunks (e.g. page-size, cache line size, etc.), the number of minimal loads to a given data level can be derived and used to write parallel code with optimized data locality.

The methods do not imply to provide better approaches to optimize instructions on one processor, but to exploit over different levels (ILP to LLP) and provide a structured way to build up physical based chunks of instruction based on their data locality. Based on these distinct instruction chunks (CBs) a generic optimization problem can be deducted to optimize (a) to the physical relations (b) - referenced as rel1:

I. the method obtains a computation graph and therefore for every level in the graph the maximal number of parallel unique chains of arbitrary sequential instructions (CBs) are known as a function of runtime-parameters.

II. a computing unit / processor has a:

i. set of latency times to retrieve data of a given size of (data size match into register, Ln-cache, memory, disc, etc.)

ii. for a given series of instructions, a minimal data size / input-bit-size S_{comp} is needed to compute a series of given instructions on a given processor or a combinational circuit.

iii. for a given series of instructions, a distinct time of computation is given on a processor / combinational circuit by the corresponding way of computing this series of sequential instructions (instruction set, processor architecture, cache and memory layout, or combinational circuit vs. sequential circuits)

This means for every gamma-graph-level (a) the number of different CBs, (b) per CB the minimal size to compute the instructions in the CB and (c) the needed "read" and "write" data sizes expressed as data transfers are known during compile time as a function of runtime-parameter. idl-points represents a gamma-node where a processor would idl e.g. see figure 80 return(0) when Γ_2 and Γ_{3b} have not the same runtime to conduct the transfer of all $S_{transfer,i}$ and calculation of problem size $S_{comp,2,and\ 3,a}$. This information is extracted in a generic form from any given code, as the information of the data relations must be in any form of runnable code.

Summarized, along a gamma-graph with initial granularity = computational graph, see II above - referenced as rel2:

(1) each level give1s the number of parallel gamma-nodes. Each CB consists of a unique chain of instructions (fig 80: Γ_2, Γ_{3a}, Γ_4)

(2) each node has for a given platform a distinct data size $S_{compute,CB}$ needed to compute the instructions (function of variable types, processor properties and available sota compile methods)

(3) a unique transfer in and out $S_{transfer,in/out}$ of the gamma-node is known

The same information is contained in the tensor notation. $S_{compute}$ and $S_{transfer}$ extracted by the method gives a) the problem size per graph-level (sum of all parallel compute sizes in parallel CBs $S_{compute} = \sum S_{compute,Cb\,i}$) b) the set of unique series of instructions runnable in parallel c) needed size to compute $S_{compute,CB,i}$ for every gamma-node. Therefore, $S_{compute,CB,i}$ defines the minimal register size (or use of higher memory hierarchies) on a given platform to compute the series of instruction in the CB and in case CBs are seen as combinational circuits, a distinct minimal area needed on a silicon wafers N. Weste, et al., "*CMOS VLSI Design, A circuits and systems perspective*", Addison-Wesley, 2011. In combination with the $S_{transfer}$ the number of loads to an available fastest memory hierarchy (memory, flip-flops, etc.) is known. These size properties are known with EV3 as a function of loop-variables and loop-header-definitions. Based on this EV1 and the method enables to bring (1), (2), (3) (see rel2) in relation to (i), (ii), (iii) (see rel1) and therefore numerical optimizable.

This builds a base for a wide range of different optimization approaches for real applications, as in real application ideal parallelism assumptions are not given.

Optimized scheduling for loop-sections – loop level parallelization

Loop-level parallelization is one of the central aspects in High-Performance computing (HPC) applications and High-Performance Technical Computing (HPTC) applications, as loop-section codes may introduce large computing demands. HPC and HPTC uses supercomputers and computer clusters to solve advanced computation problems. In particular, with the increasing complexity of integrated circuit (IC) designs at Nano-Tera scale, multi-core CPUs and multi-core GPUs have become ideal hardware platforms for emerging parallel algorithm. Today, multi-core processors are widely used across many application domains, including general-purpose, embedded, network, digital signal processing (DSP), and graphics (GPU). Core count goes up to dozens, and for specialized chips over 10,000, and in supercomputers (i.e. clusters of chips) the count can go over 10 million. To use such platforms efficiently, parallelization, in particular optimized loop-parallelization, is technically absolute essential. However, one of the technical problems of parallelization also arises from the fact, that it has become extremely challenging to leverage parallel hardware platforms at extreme scale beyond 22nm and 60GHz where complex source codes and algorithms, such as

circuit simulation, show strong data dependencies. The inventive auto-parallelization system provides, inter alia, data dependency elimination in parallelization of complex codes, such as technical circuit simulation e.g. parasitic extraction, transient simulation, and periodic-steady-state (PSS) simulation, which paves the way towards unleashing the underlying power of parallel hardware platforms. The improvement in performance gained by the use of parallel-processing systems, such as a multi-core processor, depends strongly on the level of parallelization achieved. In particular, possible gains are limited by the fraction of the parallelized code that can run in parallel simultaneously on the multiple cores or processors; this effect is described by Amdahl's law. In the best case, so-called embarrassingly parallel problems may realize speedup factors near the number of cores, or even more if the problem is split up optimized enough to fit within each processor's or core's cache(s), avoiding use of much slower main-system memory. In the prior art, most applications, are not accelerated as much even refactoring is used. The present inventive auto-parallelization system allows to achieve the possibly highest optimization, where data dependency in the column of the computations matrices vanish.

Loop-level parallelization in computer architecture is complex. The technical object of loop-level parallelization is to take out parallel tasks within the loops in order to speed up the process. The need for this parallelism, in particular, arises where data is stored in random access data structures like arrays. A program that runs in sequence will iterate over the array and perform operations on indices at a time, a parallelized code that has loop-level parallelism e.g. will use multi-task/multi-threads/multi-processes that operate on the indices at the same time or at different times. As discussed before, opportunities to exploit parallelism is mostly in application where data are stored in random access data structures. In a loop, data dependencies can be classified, as follows:

type	***example***	***notation***	***description***
True(Flow) Dependence	S1: int a, b; S2: a = 2; S3: b = a + 40;	S1 → T S2	A true dependence between S1 and S2 means that S1 writes to a location later read from by S2
Anti-Dependence	S1: int a, b=40; S2: a = b - 38;	S1 → A S2	An anti-dependence between S1 and S2

	S3: b = -1;		means that S1 reads from a locatio0n later written to by S2
Output Dependence	S1: int a, b=40; S2: a = b - 38; S3: a = 2;	S1 → O S2	An output dependence between S1 and S2 means that S1 and S2 write to the same location
Input Dependence	S1: int a, b, c=2; S2: a = c - 1; S3: b = c + 1;	S1 → I S2	An input dependence between S1 and S2 means that S1 and S2 read from the same location

Table: The table illustrates the "read" and "write" dependencies captured as well by the inventive system and method

An important distinction is "loop-carried" versus "loop-independent" dependencies. In the case of "loop-independent" dependencies, there is no dependency between the statements in each iteration, for example:

```
for (int i = 1; i < n; ++i) {
      S1: tmp = a[i];
      S2: a[i] = b[i];
      S3: b[i] = tmp;
}
```

In contrast is loop-carried dependence, for example:

```
for (int i = 1; i < n; ++i) {
      S1: a[i] = a[i-1] + 1;
}
```

For the present application, the following classification can be used: (1) DISTRIBUTED Loop: Statements can be extracted which are not dependent and can be computed in separated loops and in this way distributed;

for (int i=1; i<n; ++i) {
S1: a[i] = a[i-1] + b[i];
S2: c[i] += d[i];
}
	loop1: for (int i=1; i<n; ++i) {
	S1: a[i] = a[i-1] + b[i];
	}
	Loop2: for (int i=1; i<n;++i) {
	S2: c[i] += d[i];
	}

(2) DO-ALL parallelism (Independent multi-threading (IMT)): Statements can be extracted within loop executable independently. So all statements in the loop-core can be executed independently;

for (int i=1; i<n; ++i) {
S1: a[i] = b[i] + c[i];
}
	begin_parallelism();
	for (int i=1; i<n; ++i) {
	S1: a[i] = b[i] + c[i];
	end_parallelism();
	}
	block();

(3) DO-ACROSS parallelism (Cyclic multi-threading (CMT)): Statements, respectively calculations can be extracted that can be performed independently and running simultaneously;

for (int i=1; i<n; ++i) {
a[i] = a[i-1] + b[i] + 1;
}
	for (int i=1; i<n; ++i) {
	S1: a[i] = a[i-1] + b[i];
	post(i);
	}
	for (int i=1; i<n; ++i) {
	wait(i);
	S2: c[i] += a[i];
	}

(4) DO-PIPE parallelism (Pipelined multi-threading (PMT)): Parallelism is exploited when a loop iteration is distributed over synchronized loops.

for (int i=1; i<n; ++i) {
S1: a[i] = a[i-1] + b[i];
S2: c[i] += a[i];
}
	for (int i=1; i<n; ++i) {
	S1: a[i] = a[i-1] + b[i];
	post(i);
	}
	for (int i=1; i<n; i++) {
	wait(i);
	S2: c[i] += a[i];
	}

DISTRIBUTED Loop parallelization is the simplest one and does not have to be explained further. In DO-ALL parallelization, every iteration of the loop is executed in parallel and completely independently with no inter-thread/inter-task communication, as done in DISTRIBUTED Loop parallelization. The iterations can be assigned to threads/tasks in a round-robin fashion. The round-robin method is a scheduling method that processes queues, among other things. For example, the round-robin method can be used as a process scheduler, where it allocates limited execution resources to several competing processes or, in parallelization, processing units as processors. The round-robin procedure grants all processes successively allocation to one or more executing units for a short period during a time slot. In the technical field, this process is also called arbitration. In parallelization, round-robin can e.g. be used for load balancing of the processing units. DO-ALL parallelization and DISTRIBUTED Loop parallelization are possible only when the loop does not contain loop-carried dependencies or can be changed so that no conflicts occur between simultaneous iterations that are executing. Loops which can be parallelized by DO-ALL parallelization are likely to experience speedups since there is no overhead of inter-thread communication. However, the lack of communication also limits the applicability of this technique as many loops will not be amenable to this form of parallelization.

In DO-ACROSS parallelization, like independent multi-threading, iterations are assigned to threads/tasks in a round-robin manner. Optimization techniques described to increase parallelism in independent multi-threading loops are also available in cyclic multi-threading. In this technique, dependencies are identified by the compiler and the beginning of each loop iteration is delayed till all dependencies from previous iterations are satisfied. Like this, the parallel portion of one iteration is overlapped with the sequential portion of the subsequent iteration. As a result, it ends up in parallel execution. Once all cores have started their first iteration, this can approach linear speedup if the parallel part of the loop is very large to allow full utilization of the cores.

DO-PIPE parallelization is the way for parallelization of loops with cross-iteration dependencies. Here, the loop body is divided into a number of pipeline stages with each pipeline stage being assigned to a different core. Each iteration of the loop is then distributed across the cores with each stage of the loop being executed by the core which was assigned that pipeline stage. Each individual core only executes the code associated with the stage which was allocated to it. However, there is no consistent way in the prior art as now provided by the inventive system, to handle and automate loop-level parallelization, in particular not on auto-parallelization level.

In the compiler technology, loops are represented as different nodes in the control flow graph (CFG), e.g. for Low Level Virtual Machine (LLVM). It is to be noted, that a CFG, as used herein, is not only a graphical representation, but accurately represent the flow inside of a program unit, which is the reason that it is used in complier technology and systems. Loop-definitions and their influence on an induction variable during runtime are often depending on runtime-parameters. As the location of the change is known in the code, the "***reading***" and "***writing***" concept of the inventive system and method can be applied to array expressions in the loop-bodies. Therefore, the influence on array expressions with dependencies to loop-variables can be determine during compiling as described in above for the Basic Blocks in body/bodies of loop-sections by distinguish in different computation blocks (CBs): (1) Reading data from random-access data structure (e.g. an array); (2) computing statement; (3) Writing data back to the data structure. This is illustrated in figure 91, where a simple loop section in Control Flow with a loop body with LLVM IR code is shown and how this leads to "***read***", "***compute***" and "***write***" computation blocks (CBs) with the inventive method. With a simple example of the form a[i + Δiw] = a[i + Δir] + C with Δiw = 4 and Δir = 0 the CBs in the loop, e.g. from i=0-7, this can be represented as illustrated in figure 92.

As discussed above, the loop section can be represented by a Gamma-graph with parallel task-nodes, each with one or more combined computation blocks ($n_{||}$) and a correlated number of iterations (n_{loop}). Figure 93 illustrated such loop-sections with parallel CBs and explicit loop iterations.

In this form the method, transfers bodies of (nested-) loop sections into DISTRIBUTED or DO-ACROSS/DO-PIPE loop-parallelism setups. As discussed above, two different cases can occur when building gamma-nodes from (nested-) loop sections:

- ***case 1*** - generic CB with 1-read-write, which leads to a DISTRIBUTED loop-parallelism

- ***case 2*** - generic CB with K-read-write, which leads to a DO-ACROSS/DO-PIPE loop-parallelism

Hardware parallel processing architectures with limited resources

For the case of hardware parallel processing architectures with limited resources, there can be the case that, due to the limited resources, i.e. the number of available computation units $n_{\text{units}} < n_{||}$, the optimization step of the inventive method to schedule the gamma-nodes to the different n_{units} differs between the two cases:

- ***case 1 - 1-read-write-case***: indicated in figure 80 gamma-nodes connected with 1 edge (or CBs on one row with no entries in the transfer matrix and with the same branch node) can be combined, meaning the CBs building a chain again of a clear "read" and "write". Following the example in figure 96, the loop is represented as $n_{||}$ gamma-nodes in the level, each with a length n_{loop}. The optimization in case of limited resources is to distribute these $n_{||}$ gamma-nodes evenly. As a gamma-node consists of CBs with sequential instructions, it is possible to split a gamma-node between one of the sequential instructions, to split the gamma-node to two different units, see figure 103. On each unit n_{evenCBs} are scheduled and the remaining gamma-nodes $n_{\text{partialCBs}}$ (n_{evenCBs}, $n_{\text{partialCBs}}$) = divmod($n_{||}$, n_{units}) evenly to the units by adding transfers. This is beneficial, as long as the added latency times by the additional transfers is sufficient smaller than the corresponding distributed computations.

- ***case 2 - k-read-write-case***: The optimization is to combine computation blocks (resulting in sum up the computation and vanish of transfers) to evenly distribute the CBs to the available n_{units} units. In this case the combination of the CBs should minimize the resulting transfers.

It can be shown, that both optimization steps are under the given assumptions solvable by an analytic engine (i.e. a system being based on at least an arithmetic logic unit, a control flow in the form of conditional branching and loops, and integrated memory. Method solving a certain problem and being processable by a finite analytic engine within a finite time are also called Turing-complete. With other words, the present inventive auto-parallelization system and method provides a Turing-complete system since every auto-parallelization of any possible source code is computable with the inventive method on a computational system with respect to the class of auto-parallelization problems. This is not given for all known prior art auto-parallelization systems), which proves the novelty of the inventive system and method to parallelize loop-section during runtime, since this imposes the highest technical hurdle in prior art auto-parallelization systems. In fact, it is important to note, that the present inventive hardware-architecture specific optimized parallelization of a source code achieved by the inventive system and method does not only work for loop-level parallelization, but is generally applicable to the parallelization of source code. Under the assumption that $t_{computation}$ of a computation block (CB) is small in respect to $t_{data-transfer}$, the computation blocks within a column of the computational matrix can be evenly distributed to form even long tasks distributed to the parallel processing units/processors/cores. Still under the assumption that $t_{computation} < t_{data-transfer}$, the number of CBs do not have to be even distributable to the parallel processing units/processors/cores available in a specific hardware, since the number or CBs in a task may vary at most by the number of 1, if this embodiment variant is used. This holds when all parallel processing units/processors/cores have similar performance characteristics and the transfer latencies are of the same magnitude.

As an embodiment variant, a "*symmetrical hardware platform*" serves as basis, where the source code is to be auto-parallelized by the auto-parallelization system hardware-specific and hardware-optimized. A symmetric hardware platform has computing units (processors/cores), of which all units have approximately the same computational power and the same transfer properties between each other. This is often given, at least approximately, for the cores of multi-core CPUs. In this setting the inventive method allows to optimize the scheduling of the gamma-nodes in a Turning-complete manner, in the case the overall computing time on one unit is not faster, than the overhead to run the problem on more than one unit. This is possible as all parallel CBs have the same computation demand and therefor have the same computational effort and all computing units can solve these computations at the same time.

The inventive auto-parallelization system and method further allows to introduce a new structure how to handle (nested) loop-sections in a code in BB and to build the tasks depending on the number of processing units n_{units} of a specific hardware architecture. As analyzing all data dependencies in loops is not practical (i.e. using a brute-force method by trying all possible solutions is typically not applicable to NP-complete problems like the loop-parallelization), the inventive method is used to break or structure loop-section according to figure 94. Thus, the inventive system and method allows to form gamma-graphs solely depending on the loop-sections in a CFG with BB without unrolling and analyzing all correlated data-relations (brute-force). The inventive system and method build a generic loop-structure formed from the CBs for a loop-section.

When distributing a gamma-node of a loop-section to computing units, the following phases need to be differentiated:

(i) ***initial phase***: get data from the location (computing unit) where last writing occurred before the loop-start

(ii) ***computation phase***: compute parallel $n_{||}$ CBs on a given computing unit n_{loop} times

(iii) ***inner-loop-mapping phase***: transfer data for next loop iteration on all involved computing units

(iv) ***result phase***: when n_{loop}-iterations on all computing units are finished, data could have to be transferred back to a host

Depending on a target platform, the different sections can be shaped accordingly. E.g., for a cluster configuration, where local stored data is stored on disc and synced by the filesystem, the result-part is not necessary to explicitly applied. Another example is in a multicore environment, where data is shared in memory. There, the transfers are not explicit, but inner-loop-mapping can be used as barrier in accordance to avoid race-conditions (see above). Figure 95 illustrates the 4 phases for an example, where n_{units} = 4.

As $n_{||}$ is a function of the inherent read and write constraints in a statement of the form $a[i + \Delta iw] = a[i + \Delta ir]$, it is possible for the inventive method to derive $n_{||}$ by analyzing the array expression in the loop-body and find a minimal read-write distance ΔI_{rw}. This must be possible, as array indices must be in the range of the array definition and must be of positive natural numbers otherwise access on the array is not possible. The change of the indices is analytically derivable during static-compiling as the step, when a loop-variable changes (is written) and must be retrievable in any code. This position in code defines when this information is available during runtime and therefore when depending parallel opportunities can be exploited. Knowing $n_{||}$ the iterable-loop variable-limit n_{loop} can be derived from the loop-definition, respectively during compile-time the numeric relation between run-time parameters and $n_{||}$ and n_{loop} can be extracted. The system can analyze the reading and writing of an array-expression during static compiling and generate a gamma-graph as a function of $n_{||}$ (which can be a function of loop-parameter) and n_{loop} resulting in distributed loops with n_{loop} iterations, see figure 93. This can be done without resolving every data-dependency. $n_{||}$ and n_{loop} are in most cases depending on loop-parameters and therefore not known until runtime. If all parameters are known, the scheduling of the gamma-nodes can be optimized analytically during compile-time. In case the parameters are runtime-depending, on the defined symmetrical platforms the inventive system and method can be used to solve scheduling analytically during runtime. To optimize the gamma-nodes to computing units, there are two cases.

Case 1: *Generic CB with 1-read-write*

In the case there is only one read-write dependency in the array expression(s), the corresponding gamma-node has the form in Figure 96 with the example of $a[i + 3] = f(a[i])$. There is no inner-loop mapping phase, and the inventive system 1 results in DISTRIBUTED-parallelism. The computational load $n_{loop,\Gamma i}$ for each gamma-node is depending on the loop-parameters and therefore in most cases runtime-dependent. But $S_{compute\Gamma_i}$ and $S_{transfer\Gamma_i \to \Gamma_j}$ are computable with simple arithmetic processes: the $n_{loop,dist}$, $n_{loop,reminder} = divmod(n_{twt}, n_{||})$ where each gamma-node receives CBs - grouped together in one gamma-node - as CB connected with 1 edge (= transfer) and they build an arbitrary sequence of sequential instructions $n_{loop,i}$= $n_{loop,dist} + n_{loop,reminder,j}$, j ∈ $[n_{loop,reminder}]$ otherwise j=0.

Case 2*: Generic CB with K-read-write*

In the case there are K reads to the write dependency in the model CB, the corresponding gamma-node has the generic form illustrated according to the example in figure 97 with the example of *a[i+5] = f(a[i], a[i-1])*. There is an inner-loop mapping phase and the inventive system results in DO-ACROSS/DO-PIPE loop-parallelism. Thus, the inventive system 1 and method derives:

- Maximal parallel CBs per iterable loop step: $n_{||} = f(\Delta I_{rw}) = min(\Delta iw - \Delta ir) = \Delta I_{rw} = 5$
- Transfers between each CB to the next iteration in case of $n_{||} = n_{units}$: $n_{transfers} = 1$
- Initial granularity: $G_0 = \frac{\Delta t_{comp}}{\Delta t_{comm}} \sim \frac{S_{compute\Gamma_i}}{S_{transfer\Gamma_i \to \Gamma_j}}$

With this, $S_{compute\Gamma_i}$ and $S_{transfer\Gamma_i \to \Gamma_j}$ per gamma-graph can be deducted as a function from loop-parameters by analyzing the code. Each read has in the next iteration a transfer from other CBs, except one write->read dependency, which can vanish by $n_{||} = \Delta iw - \Delta ir$ as it is computed on the same computing unit. In case of several reads, $n_{||}$ is the minimal distance of all read-write-differences, as illustrated in figure 98.

Distribute load in computing section on symmetrical parallel processing machines

The runtime to execute one iteration of the iterable-loop with n_{loops} is then minimal, when all uniform computing units have an equal amount of CBs. This means all parallel computation blocks ($n_{||}$) can then be distributed according to a relation, where G is a graph which consists of a set of independent nodes with weights $\{w_1, \ldots, w_n\}$, and k is the number of available processors. Then

$$CT_{PS}(G,k) = max\left\{w_1, \ldots, w_n, \sum_{i=1}^{n} w_i/k\right\}$$

For the above scheduling of real time tasks on multiprocessor systems, it is clear that the above given computation time cannot be improved upon by any system since the schedule must be at least as long as the largest task and cannot be more efficient than to keep all the processors continuously busy. The number of parallel CBs $n_{||}$ have an initial-granularity

$$G_0 = \frac{\Delta t_{comp}}{\Delta t_{comm}}$$

with Δt_{comp} and needed transfers with Δt_{comm} in the inner-loop mapping. Distributing the $n_{||}$ to available number of same units n_{units} is a simple analytical correlation. By combining the CBs, the inventive system is able to generate a task-graph with a corresponding G_{nunits}, where three cases can be differentiated:

- $n_{||} > n_{\text{units}}$: Optimize and combine parallel CBs to get $G('()"*$
- $n_{||} = n_{\text{units}}$: there are exact resources available and gamma-node with GC can be used
- $n_{||} < n_{\text{units}}$: there are more resources available than can be used → use $n_{\text{units}} = n_{||}$ and run with G_0.

The optimization steps, performed by the inventive system 1, are then to distribute the parallel CBs following the relation (see above) to generate an optimal task-granularity G_{nunits}:

(a) Distribute $n_{||}$ CBs to n_{units} to get approximately an equally distributed Δt_{comp} on each computing unit

(b) minimizing transfers between the gamma-node with granularity G_{nunits} to minimize Δt_{comp}

It is to be noted that step (a) is a step, which can be performed arithmetically by the inventive system 1, in contrast to step (b), where the system 1 must optimize the inner-loop-mapping. Below, it is shown that this mapping can also be performed analytically by the inventive system by combining only CBs which minimizing the transfers as they vanish when the CBs are combined on one computing unit.

Inner-loop-mapping optimization for G_{nunits}

When combining parallel CBs to gamma-nodes (with a resulting granularity G_{nunits}, transfers on the same unit vanish and computations sum up, see figure 99. With a very simple illustration, it can be shown that only combinations with one of the read-write distances is minimizing the number of transfers between gamma-nodes in each iteration of the iterable-loop (n_{loop}). This can be summarized as illustrated by figure 100 by:

- Reading accesses (right site of a statement or load in an IR) and the number of reads is extractable and the number of read-accesses is denoted as n_{shifts}.
- Number of transfers per CB: $n_{transfers} = n_{shifts} - 1$ as 1 write-read can be vanished by optimization to run on same unit in the next iteration.
- Size of transfer per CB with S_T is the size of one transfer: $S_{transfer} = n_{transfers} \cdot S_T$
- For the example in figure 101 this results in $S_{transfer} = 2 \cdot S_T$
- Combined CBs not along read-shifts: $S_{transfer,combo}(n_{comb}) = n_{comb} \cdot n_{transfers} \cdot S_T$
- Combined CBs along read-shifts: $S_{transfer,combo}(n_{comb}) = n_{transfers} \cdot S_T = const$

As illustrated in figure 101, depending on $n_{transfers}$ any combination will lead to a fixed $S_{transfer,combo}$, as the reading is in the iteration from the same computation unit when combining more than n_{comb} CBs to one gamma-node. Illustrating with the statement $a[i][k] = a[i-2][k] + a[i+5][k]$, some interesting properties can be observed (cf. figure 102):

- With T, one transfer of the size S_T can be labeled, which is depending on the array-type 'a'
- With $read - shift = -2$ origin from the expression part ($[i-2][k]$) and $read - shift = +5$ as a result of the expression part ($a[i+5][k]$), the iteration distance with a loop-iteration-step-size of 1 is $n_{comb,const} = max(read - shift) - abs(min(read - shift)) = 7$

- There evolve 2 boundaries: boundary A and boundary B, when combining these to one gamma-node based on $n_{\text{comb,const}}$.
- Combining more than $n_{\text{comb,const}}$ does not increase/decrease the number of transfers.
    - Transfer over boundary A: min(read-shift): $2 \cdot S_T$
    - Transfer over boundary B: max(read-shift): $5 \cdot S_T$
    - Total Transfer from Gamma-node to Gamma-node at boundary A and boundary B: $S_{T,\text{const}} = (2 + 5) \cdot S_T = 7 \cdot S_T$
- Gamma-node with one CB has two transfers: $S_{T,1CB} = 2 \cdot S_T$

When combining n_{comb} CBs with $n_{\text{comb}} > n_{\text{comb,const}}$ the number of transfers get constant, $S_{T,\text{const}}$, as illustrated by figure 103. This effect is visible also in the computations. Finding the combinations for the range $1 < n_{\text{comb}} > n_{\text{comb,const}}$ in Figure 101, the CBs must be combined according to the write-read distances, what leads to a linear increase of transfers.

Scheduling for case 1 - generic CB with 1-read-write

If there are limited resources n_{units} in the 1-read-write cases, it can happen that there is a partial left when distributing gamma-nodes to n_{units}: $(n_{\text{full}}\ n_{\text{partial}})$ = divmod($n_{||}$, n_{units}). In this case the n_{partial} gamma-nodes have to be split evenly to the available n_{units}, which is illustrated by figure 103. This can be done by adding additional transfers as a function of $(n_{\text{full}}\ n_{\text{partial}})$. Adding additional splits makes only sense as long as distribution of the partial CBs by adding additional transfer is not longer than without splitting. As the gamma-nodes (one or more CBs) contain arbitrary sequential instructions, the place where to split has no significant influence on Δt_{comp} and can be defined by *partial*/n_{units}. As an example, in case of a[i + 3] = $a[i]$ on n_{units} = 2 this leads to (1,1) = divmod(3,2) leads to adding one additional transfer by splitting one gamma-node.

Scheduling for case 2 - generic CB with K-read-write

In case the resources are limited and $(n_{\text{full}}\ n_{\text{partial}})$ = divmod($n_{||}$, n_{units})n_{partial} = 0, then the computational effort can be balanced out with minimizing the transfers. If

there is a partial result $n_{partial} > 0$ it is not feasible to schedule in the form all units are balanced out perfectly. This will lead to idle of the other units, but there is no possibility to balancing this inequality out, as all units have the same performances, transfers have all the same properties and the read-shifts does not make it possible to reduce any transfer.

Gaps in nested loops

Gaps in iterations occur when start value and/or iteration step size is not equal 1 in (nested-) loops and a read (or write) is from/to a gap. That means in the data structure, some values are never written within the loop, or some data is being read from CBs before the loop in each iteration step (e.g., boundary condition in a 2d heat equation). Figure 104 structured data index and resulting gaps when not iterated over the whole array.

Mapping and indexing for Gamma-nodes

There are different approaches to implement the mapping. One is to generate a global index for the unrolled loop [0, $n_{||}$] that can be generated for all nested loops start-, end- and inter-step-size marked as gaps. Then the number of combined CBs per Gamma-node can be used to map the gamma-nodes to the global index and a local-data-index can be computed. By getting unique elements in each gamma-node the communication links for each phase comprise:

- initialism,
- boundary-condition (reading from gaps during compute phase)
- inner-loop-mapping
- result

They can easily be found by computable array-arithmetic steps. Other options are to use bit-arrays or computing the global indices at the boundary A and B, see figure 102 depending on the read-shifts for each gamma-node, as illustrated by figure 103. Another approach is to compute the reduced iteration steps n_{loop} for each computing unit as a function of the runtime parameter. This is practicable during runtime and therefor makes it feasible to use the method to optimize loop-section on

symmetric parallel machines during runtime based on a model build during compile-time.

Further embodiment variants and applications of the inventive system

As illustrated by figure 106 known technical problems in the field of parallel processing can be solved by means of the inventive system and method. In particular, figure 106 illustrates the different embodiments EV1 to EV5 of the inventive system 1. EV1 of figure 106 shows the most fundamental embodiment variant and the basis of all other embodiment variant of the inventive system 1 and method, where EV1 provides auto-parallelization of code by optimizing the overall latency time to a minimum. In case there are limits by the number of parallel processors (single-core 2103 and/or multi-core 2102) available, (ii) the latency time for memory access and data transfer to and from a processor 2102/2103 to the processing time needed to compute a computation block node 333 on a processor 2102/2013 is not small, the embodiment variant EV1 is based on some fundamental assumptions, which are, inter alia, To reduce the row dimensionality to the number of the limited resource n_{units} the rows can be combined. This form of optimization is bounded with the assumptions (i) any transfer introducing a significant latency time compared when a transfer in the transfer matrix vanishes $\Delta t_{transfer} \gg \Delta t_{transfer} \approx 0$. (ii) variations between transfers are small $\Delta t_{transfer,i} \cong \Delta t_{transfer,j}$, meaning all transfers have approximate equal latency times. Assuming (i) and (ii), the maximal row-dimension in the compute and transfer matrix define the combinatorial complexity of the system (several CB per BB). This can lead to potential limits for the combinatorial complexity, especially for the loop-sections and / or if (iii) the units have similar computing performances and transfer latencies among themselves is not given. Combined with EV3 the possibility is introduced to optimize for a generic code to a (not-) symmetrical parallel machine. The following is a non-conclusive list of optimization options the tensor / gamma-graph. For each option it is annotated how to estimate compute and transfer times:

I. For EV3 / loop-sections on symmetrical platforms: the optimization step is to distribute parallel $n_{||}$ gamma-nodes to available units evenly. There is no need to know of specific compute / transfer times to distribute $n_{||}$ to parallel units n_{units}. For distributing on a

multicore with cache-latency awareness, the access latency to shared memory hierarchies (cache-levels and memory) as a function of data size for each n_{units} must be known. This is possible, as all $n_{||}$ parallel CBs have the same properties in the computation and transfer matrix cells. This means each CB has the same read, write and compute properties. Therefore, (i) a distinct chain of sequential instructions is known for each CB (ii) corresponding data size needed to compute this set of instructions with a given processor is known. EV3 comprise the process of building tasks out of computation block node 333 belonging to the same column in the computational matrix, i.e. executable in parallel due to the available data. Building tasks allows for a processor-architecture specific and/or system-architecture specific optimized auto-parallelization where the number of processors 2102/2103, the performance of the different processors/cores 2102/2103 and/or processor units 21, the differences in size and response time of the different memory units 22, in particular the different processor registers 2211 and/or processor caches 2212 and/or RAM units 2213.

II. Following Wall, David W., "*Limits of instruction-level parallelism.*" Proceedings of the fourth international conference on Architectural support for programming languages and operating systems, 1991 the ILP dimensionality in basic blocks is around 3-4 on average and this is a bounded combinatorial problem when assuming the fundamental assumptions in EV1. By applying EV3 and distributing to all available units n_{units}, the restriction for loop-sections can be solved. When all units are used per loop-section, it makes it possible to optimize the other non-loop sections. This reduces the combinatorial complexity to schedule the remaining Basic Blocks, respectively the row dimensionality of the computation and transfer matrix without the loop-sections. To approximate the runtime of CBs, either a model based on relative computing times from tabled cycles and access latencies to memory hierarchies can be used or the different row combinations can be profiled.

III. by traversing the gamma-graph: starting from a parent gamma-node, continuously combining children-nodes till a target granularity is reached. Then compose from the combined nodes a task 36 resulting in a task graph forming equally length tasks. This results in a task graph with a fixed granularity and therefore optimizing the scheduling of targeted for a given hardware granularity $G = \Delta t_{compute}/\Delta t_{transfer}$. The granularity can be estimated by # instructions, cumulative cycles of the instructions and corresponding data loads of the combined CBs. In this case only the relative computational length between different CBs is important obtainable by tabled instruction values, see A. Fog, "*Instruction tables*", Technical University of Denmark, 2022. This reduces the effort to scheduling tasks as they can be distributed evenly as they have an optimized task-granularity for a given platform. This is a bounded problem, but the approach could miss some overall optimization opportunities.

IV. In context of EV2 it is possible to associate a fixed value transfer time (TT) by a given input to the compiler for all transfers (edges in the gamma-graph). In this form the gamma-graph can be formed by combining parallel gamma-nodes till their approximated computation time (e.g. derived by tabled values) is beneficial over the computation in parallel than in serial. The implementability of this approach in auto-parallelization was proven in a pre-study by the Institute Reconfigurable & Embedded Digital Systems (REDS Institute) in Lausanne, Switzerland. It could be shown that it is possible to build depending on a given latency time TT adapt a gamma-graph to generate a latency adapted task graph with corresponding task granularity. For this pre-study the assumptions were, that all parameters are known (without the use of E3).

V. Using the gamma-graph as an input to form a (Mixed) Integer Linear Programing Problem: Building a linear optimizable set of equation, where the gamma-graph EV2 represents a similar complexity as the PCGmTSP. To estimate for the different m gamma-node-groups either tabled or profiled approaches can be chosen. Combined

with EV 3 loop-section can be associated with distinct number of used units (m), depending on the parallel groups per gamma-graph level.

Finally, the embodiment variant EV4 of figure 106 (see also figure 110) shows the inventive system used for optimization of integrated circuit (IC) or chip design solving electronics engineering technical problem if IC design by encompassing logic and circuit design required to design integrated circuits, or ICs. In particular, the IC design provided by the embodiment variant EV4 provides digital IC design which can be used to produce components such as microprocessors, in particular multi-core microprocessors, field-programmable gate arrays (FPGAs), memories (cache, RAM, ROM, and flash architectures) and digital application-specific integrated circuits (ASICs). ICs comprise miniaturized electronic components built into an electrical network on a monolithic semiconductor substrate by photolithography. The digital design achieved by the embodiment variant EV4 provides highly optimized IC architecture in regard to logical correctness, maximizing circuit density, and placing circuits so that clock and timing signals are routed most efficiently.

Below, some application of the inventive system 1 and method are discussed in more detail.

(i) Auto-parallelization of source code for generating Fibonacci series

Figure 75 shows exemplary the construction of the CBs out of instructions (CMP, SUB, ADD) visible in figure 76 and as a gamma-graph in figure 81. This is the representation for a recursive call f(3) of the code in figure 75. A similar graph can be retrieved when applying the method on the loop-implementation of the Fibonacci series.

To generate and compute the Fibonacci series, different parallel-processing implementations with different performance properties are known. In the following, it is illustrated how the invention is applied to an implementation using a) recursive functions calls and b) loops. Both implementations show different performances. For understanding the processing problem, reference can, for example, be taken to https://www.geeksforgeeks.org/program-for-nth-fibonacci-number/. Below, there is an example of a processing (source) code generating Fibonacci series using recursion.

```
//Fibonacci Series using Recursion
#include<studio.h>
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}

int main ()
{
    int n = 9;
    printf("%d", fib(n));
    getchar();
    return 0;
}
```

The parsing the function code of the fib(n) declaration results in a pseudo token language, with the function code of the fib(n), given by

```
int fib(n)
{
    if (n <= 1)
        return n;
    return fib(n-1) + fib(n-2);
}
```

and the pseudo token language given by the following table 1:

n, -, -, cfuncdef	cfuncdef (gid1, -, -)
n, 1, cmpV, <=	<= (gid1, 1, gid2)
cmpV, -, -, if	if (gid2, -, -)
n, -, -, return	return (gid1, -, -)
t1, n, 1, -	- (gid3, gid1, 1)
t1, 'fib', t2, callf	callf(gid3, tcbid, gid4)
t3, n, 2, -	- (gid5, gid1, 2)
t3, 'fib', t4, callf	callf(gid5, tcbid, gid6)
t2, t4, t5, +	- (gid4, gid6, gid7)
t5, -, -, return	return(gid7, -, -)

-, -, -, efuncdef	efuncdef(-, -, -)

(table 1)

Figures 41 and 42 show schematically the computation block nodes (CB) with the operation and corresponding data nodes (similar to the token in table 1 above). The icons indicates a transfer to the location in the computation block node (start- or end-point of the computation block node). The recursive call of the functions results in additional transfers, as can be seen in figure 42.

The next step is to number the computation block nodes depending on their call-position in the code. This results in a pseudo graph like schematically represented in figure 43, which in return results in the computation and transfer matrix as shown in figures 44 and 45.

By combining start- and end-communication cells in the transfer matrix and eliminating empty cells in the computation matrix and bring them back to different code segments, results in Fig. 46. Based on this code segments a code can be generated directly (as a compiler) or transferred back to code and then using a SOTA compiler to generate machine code (as a transpiler).

To show how the inventive system and method brings advantages to the compiling of a recursive implementation according to the Fibonacci series, the following passages explain how the inventive method maps and/or optimize the code in a more concurrent solution than the input code. As indicated to optimize the code it is the combination between the rows, the graph is shown with the combination of the cbn in the branch node marked as branch2b (see figure 43) and combine them, see figure 48. Calling a function is in the method placing the computation block nodes to the correct location in the matrices, respectively to apply the corresponding transfers as indicated in figure 47. With this in mind, the recursive call of the function can be seen with the inventive method as a transfer and 'read' and 'write' of the function parameters and result variables in a return statement, figure 48.

According to the example of fib(4), figure 49 shows step-by-step how additional computational block nodes and the reducing of the transfers (as all is on one computation-chain) results in a more optimized source code.

The depth of this chain depends directly on the number n in fib(n). As recursive calls can be very simply detected in the code, it is easy to not implement a recursive call in full dimension in a final application. To make it understandable, figure 50 shows this with some simplification. Step 4 shows the cbns for a call with 'n=4'.

The next step in figure 51 shows the transfer, which will be occur (transport the information of the last "write" to a data node to the place where the "read" of a data node occurs and remember this information in the corresponding computation block nodes). As all computations are on one chain, the transfers will vanish, as can be seen in figure 52. When resolving every step, this leads to a program of the form in figure 53.

Bring this using the matrix representation back to code, it can be seen as a result the code below. This results in a more efficient code, than the original fib(n=4) implemented by table 1, above. Respectively compiling this code with a SOTA compiler will result in a better optimized code, than without applying the method.

```
f(n) {
    if (n <= 1):
        return n;
    t1 = n-1
    t2 = f(t1)
    t3 = n-2
    t4 = f(t3)
    t5 = t2+t4
    return t5
}

main() {
    r1w = f(1)
    r0w = f(0)
    r2w = r1w+r0w
    r3w = r2w+r1w
    r4w = r3w+r2w
}
```

The recursive call can also be interpreted like an array operation applying the inventive method, as it is transferring an information (function parameter param[i]) to the corresponding cbns in the branch node of the function declaration and then transfer back the return value(s) to a[i], see figure 54. This would lead to a perspective seen also in partial differential equation. This form of the implementation of the Fibonacci series will be references more below. However, as a next step, the next

paragraphs will show the handling of Partial Differential Equation, which mostly result in nested loops and heavy use of array operations.

(ii) Partial Differential Equation (PDE)

With the rule to place an operation node in dependency of the "reading" and "writing" pattern and resolving not clear dependencies by a transfer, according to the patent application, it is possible to derive a compute- and communication-model for a freely given discretized implementation of a PDE. The PDE of a 2D heat equation will be used herein, with the 2D heat equation given by:

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

Using a finite difference scheme to discretization:

$$u_{i,j}^{k+1} = \gamma\left(u_{i+1,j}^{k} + u_{i-1,j}^{k} + u_{i,j+1}^{k} + u_{i,j-1}^{k} - 4u_{i,j}^{k}\right) + u_{i,j}^{k}$$

Figure 55 shows part of an implementation in python. Each entry in an array is a data node. Reading from an array index is in the method an operation node with data node of the index and the base address of the array. Array operations can be seen as operation nodes with corresponding data nodes, see figure 56.

For the example of a loop array with an array operation a[i+Δiw] = a[i] figure 57 shows the corresponding computation block nodes. With this in mind, the initial block (figure 55) can be expressed in detail in the graph like in figure 58. All computations in the computation block (see figure 55) happen in the j-loop. By using a 1D array notation and applying the fundamental rules of the method, a schematic form for the computation block node in the loop is derived as indicated in figure 59.

An array read (e.g. u[k][i+1][j]) would create a computation block node and the meta-data to transfer of the value to this index would be added to both cbns, the one which the "reading" node and the one where the last time this data node was written (e.g. at a[k+1][i][j]). This leads to the fact, that a statement such as the array operations in the j-loop (figure 55), result in 5 computation block nodes, representing

"reading" operations of the array, then a computation block node which computes the arithmetic solution and then a computation block nodes with the write to the array at the position [k+1][i][j]. This form of representation is a schematic perspective to show more clearly how the method take such array operations into account. This leads to a situation indicated in figure 60.

One of the most fundamental principles of the inventive method is to find the last operation node A, which "writes" to a data node B, and then place a new operation node C, which "reads" from B, after the operation node A. If it is not a clear 0- or 1-dependency, add a transfer to the computation block nodes containing the operation node A, respectively C. Therefore, a[i1] = a[i2] is "reading" of data node with base-address 'a' and index 'i2' and "writing" to data node with base-address 'a' and index 'i1', see figure 56. Each loop creates therefore new computation block nodes with a "reading" or "writing" operation node and the corresponding transfers. It can be derived the following scheme, figure 61.

As a loop can be seen as in figure 61, showing that every loop passage is a new computation block node on a new computation-chain (row in matrix). As long as there are no transfers, these computation block nodes will be at the same time (results in the same block or segment number in the numbering process step), as empty computation block nodes and entries in the transfer matrix vanish. If a transfer occurs, the cbns-numbers would be different, and they would not be computable in the same step. An offset in the index of the "reading", respectively "writing" operation can lead to a transfer between the computation block nodes containing the "reading", respectively "writing" operation nodes as can be seen in figure 62.

The dependencies in "reading" and "writing" of the indices in a loop leads to the conclusions in figure 63. If ΔI is smaller than 0, shows that the loop can be enrolled on as many computation block nodes as the loop is long. Long meaning the number of iterations defined by the start value 'i0' and the maximal value 'Ai' as well as the increment value 'Bi' in a loop definition, e.g. 'for i=i0; i <> Ai; Bi'. if ΔI is larger than 0, only ΔI computation block nodes can be run in parallel. If less than this number are used, transfer will occur. If the loop is resolved on more than ΔI units, the computation block nodes would not be parallel (meaning have the same number in the flow graph). Therefore, ΔI is a unique number to decide how a "reading", meaning using the

information in an array at an index, but not change it, can be distributed on different computation units.

This has some important consequences: It is possible to extract from nested loops one single dimension to decide for each "reading" to an array, which of these readings lead to a "writing" within the loop, and which values are "readings" from data nodes before the loop. Therefore, for each "reading" it can be derived if there is a transfer to the computation block node (=computation-chain) of the "writing" part. This can be implemented as a model and leads to a new, pretty generic perspective on arrays in nested loops. Furthermore, it can be derived which of the nested loops can be resolved, means in the method to be computed parallel, because there is no transfer between a "reading" and a "writing" data node, according to the concept in figure 63. For the 2D heat equation with Central Difference Scheme, this results in the dependencies shown in figure 64. This means, that the i and j loop can be resolved, as ΔI are always smaller or equal than the number of iteration in the i- and j-loop, but not in the k-loop. Meaning the i- and j-loop can be resolved in the method. This means the "reading" and "writing" in the array will be distributed in computation block nodes, which can run parallel, but the k-loop has to be iterative. And after each k-loop transfers between the units will occur. At this point, keep in mind, that the method eliminates transfers in the mapping / optimization step, if they are on the same unit.

In most implementation to solve a PDE, computations occur only on a subset of an array (e.g. by handling boundary conditions differentially), what makes this step a bit cumbersome. With the example of two nested loops, for each loop-nest a rule can be derived to handle the gaps and the consequence to the "transfers" in the proposed model, as shown in figure 65. The const values are values from computations or definitions of operations before the loops in the "computation block" (see figure 55). Incorporating the gaps occurring by the looping over a subset of the array, the transfer pattern of a discretized equation can be derived and a model for the transfers, depending on the gap-sizes, sizes of the loops (= in this case meshes) results and represents the computation block nodes of the method, as in figure 66. Showing with a very small example of nX=5 and nY=4, this leads to the computation and transfer matrices as in figure 67. In the transfer matrix the arrows '->' indicate getting information and '<-' indicating sending data to other computation block nodes.

The inventive method brings out all the necessary transfers/communication between the grid elements in the grid needed for the discretized PDE. Figure 68 shows the occurring transfers between the units, as each unit is represented by one row in the matrices. In figure 68 it can be seen an extract of the transfer- and the computation-matrix, with dark grey the "receiving" side of a transfer and light grey the "sending" part of the transfer. It is important to note at this point, that this does NOT mean it has to be sent and received, it also can be e.g. shared by a shared memory segment and guarded by a barrier or lock or vanish as it is shared by cache and therefore handled by a CPU - it depends on the used transfer/communication mechanisms. This can be transferred to a model, respectively leads to a code of the form as shown in figure 69. It has to be noted, that the computations between p0 und p1 evolve, because the method is defined to add transfer from the "written" data nodes to the "reading" data nodes and as the array operations in the nested loop start with a "reading" operation node, this situation happens.

Figure 70 shows how the transfers can be used to create a time model. Grey are the meta values (e.g. known from variable type definitions or similar), which also can be used to map/optimize the matrices to a given hardware infrastructure.

3 procs → (1,2),(3,4),(5,6)

$$\Delta t = \begin{pmatrix} \sout{2 \to 1} & 3 \to 2 \\ 0 & 5 \to 4 \\ \sout{6 \to 5} & 0 \end{pmatrix} \cdot \Delta t_{trans} + \begin{pmatrix} 0 & \sout{2 \to 1} \\ 2 \to 3 & 0 \\ 4 \to 5 & \sout{5 \to 6} \end{pmatrix} \cdot \Delta t_{trans} + \begin{pmatrix} 4 \to 1 & 5 \to 2 \\ 0 & 6 \to 3 \\ 0 & 0 \end{pmatrix} \cdot \Delta t_{trans}$$

$$+ \begin{pmatrix} 0 & 0 \\ 0 & 1 \to 4 \\ 2 \to 5 & 3 \to 6 \end{pmatrix} \cdot \Delta t_{trans} + \begin{pmatrix} \Gamma & \Gamma \\ \Gamma & \Gamma \\ \Gamma & \Gamma \end{pmatrix} \cdot \Delta t_{latency}$$

$$\Delta t = \begin{pmatrix} 3 \\ 4 \\ 3 \end{pmatrix} \cdot \Delta t_{trans} + \begin{pmatrix} 2 * \Gamma \\ 2 * \Gamma \\ 2 * \Gamma \end{pmatrix} \cdot \Delta t_{latency} \qquad cache_size = \begin{pmatrix} 8 \\ 9 \\ 8 \end{pmatrix} float_size$$

$$transfer_size = \begin{pmatrix} 3 \\ 4 \\ 3 \end{pmatrix} float_size$$

These values are not used in this example. To illustrate possible optimizing steps, a very simple model is shown fully resolved in the following Figure 71, where a combination of row (1,2), (3,4) and (5,6) resulting in 3 units computing:

a combination of row (1,2,3) and (4,5,6) resulting in 2 units computing:

2 procs → (1,2,3),(4,5,6)

$$\Delta t = \begin{pmatrix} \sout{2 \to 1} & \sout{3 \to 2} & 0 \\ \sout{5 \to 4} & \sout{6 \to 4} & 0 \end{pmatrix} \cdot \Delta t_{trans} + \begin{pmatrix} 0 & \sout{1 \to 2} & \sout{2 \to 3} \\ 0 & \sout{4 \to 5} & \sout{5 \to 6} \end{pmatrix} \cdot \Delta t_{trans} + \begin{pmatrix} 4 \to 1 & 5 \to 2 & 6 \to 3 \\ 0 & 0 & 0 \end{pmatrix}$$

$$\cdot \Delta t_{trans} + \begin{pmatrix} 0 & 0 & 0 \\ 1 \to 4 & 2 \to 5 & 3 \to 6 \end{pmatrix} \cdot \Delta t_{trans} + \begin{pmatrix} \Gamma & \Gamma & \Gamma \\ \Gamma & \Gamma & \Gamma \end{pmatrix} \cdot \Delta t_{latency}$$

$$\Delta t = \begin{pmatrix} 3 \\ 3 \end{pmatrix} \cdot \Delta t_{trans} + \begin{pmatrix} 3 * \Gamma \\ 3 * \Gamma \end{pmatrix} \cdot \Delta t_{latency} \qquad transfer_size = \begin{pmatrix} 3 \\ 3 \end{pmatrix} float_size$$

$$cache_size = \binom{11}{11} float_size$$

and a combination of all rows (resulting in a 1 unit computing without any transfers):

1 procs → (1,2,3,4,5,6)

$$\Delta t = (0) \cdot \Delta t_{trans} + (6 * \Gamma) \cdot \Delta t_{latency} \qquad transfer_size = (0) float_size$$

$$cache_size = (16) float_size$$

These steps show exemplary how the method creates different ratios between computing and communications and how depending on the combination the transfers are reduced and the computations per unit are getting higher. To make this more illustrative, a very simple model of computing and communicating can be used and some "real values" for the times assumed. Two generation of Intel CPUs with according to performance values for their floating point capabilities per cycle (FP) and their frequency leads to a power value for these to compute floating point arithmetic (like Γ in the model). A P5 FP32: 0.5 and a Haswell FP32: 32 can e.g. be used with a frequency for the cycles of 66MHz and 3.5GHz and for addition and multiplication of a float value 4 cycles. To get a value for the Δtransfer, representing the communication latency in the network (the author knows that there were made a lot of assumptions and the latency is not the only key value for a network), two types of latencies are used: 1ns for an infiniband type and 50ns for a cache latency. The behavior in figure 72 can be derived for a grid of nX=2048 and nY=1024 by Δt = number of combined cbn * cpu+number of transfers*network latency.

It is always straight forward to get code back from the method, respectively the two matrices, adapted to the available IPC options. This generic form of optimizing code brings a novelty to the field of automatic code parallelization. For example, the code can be optimal mapped e.g. to an MPI-cluster infrastructure or with some more fine grained tuning applying a Hybrid approach by combining MPI (shared memory) and threading locally on nodes. By different combinations (e.g. combine the first x cbns in i-direction and the y cbns in j-direction) it is possible to get an optimal ratio between "computation" and "communication" for any given discretized PDE but depending on the available hardware infrastructure. This can be done without any by-hand interaction as the necessary properties can also be tested, or computed for an available hardware infrastructure, solving real technical problems of the prior art.

(iii) Pointer disambiguation

Pointer disambiguation is also a technical problem not solved completely (see e.g. *Runtime pointer disambiguation* by P. Alves). Applying the inventive system and method, it can be shown, that the inventive methodology technically resolves the disambiguation occurring by passing function parameters as pointers, because it takes the pointer as an information and this disambiguation will be resolved in the step of the transfer vanishing, as illustrated by figure 74.

(iv) Fibonacci using loop

The Fibonacci source can also be implemented using loops.

```
//Fibonacci Series using Dynamic Programming
#include<studio.h>

int fib(n)
{
    /* Declare an array to strore Fibonacci numbers. */
    int f[n+2];    // 1 extra to handle case, n=0
    int 1;

    /* 0th and 1st number to the series are 0 and 1 */
    f[0] = 0;
    f[1] = 1;

    for (i=2; i<=n; i++)
    {
        /* Add the previous 2 numbers in the series and store it */
        f[i] = f[i-1] + f[i-2];
    }
    return f[n];
}

int main()
{
    int n=9;
    printf("%d", fib(n));
    getchar();
    return 0;
}
```

According to the example with the 2D heat equation, this leads to the result in figure 73, and, in return, results in a source code of the form:

```
main() {
    a1 = f(1);
    b1 = f(0)
    for (i=0; i<=(n-1)/2; i++) {
        c1 = a1+b1
        a2= c1
        b2 = b1
        c2 = a2+b2
        a1 = c2
        b1 = c1
    }
}
```

This has the similar performance properties as the result the code of figure 53, shown above.

To proof of actual efficiency of the inventive system and to benchmark the performance of the inventive system, the above-mentioned recursive Fibonacci example can be used, which, in the technical field of auto-parallelization systems, is often used as proof-of-function and benchmark study (see e.g. V. Sarkar "Fundamentals of parallel programming module 1 parallelism", p. 60ff). Based on the simple example of the recursive implementation to generate Fibonacci numbers, it is possible to show "if", "how", and "how good" an auto-parallelization system and compiler is able to optimize a code for a given platform's transfer-time and processing-time. Figure 75 shows the recursive implementation, the LLVM IR in Basic Block form and how the system handles the grouping in the method. Figure 75 shows LLVM IR and grouping of instructions in computation blocks (CBs).

Two simple strategies are discussed in the following:

a) ***Optimize with latency times***: calculate all approximated runtimes for combinations for a given number of units. This references to the II. of the listed optimization strategies.

b) ***Optimize for different transfer times TT***: form different task graphs as a function of a given transfer time (TT). The different TT representing hardware platforms with different transfer latency times. This references to the III. of the mentioned listed strategies.

Optimization approach a): Optimization with latency times

To demonstrate the effect of the method along a well-known and studied example, the recursive implementation of the Fibonacci series can e.g. be used, see figure 75.

$$\begin{matrix}\Delta t_{end,u1}\\ \Delta t_{end,u2}\\ \Delta t_{end,u3}\end{matrix} = \Delta t_{c1} + \Delta t_{c7} + \begin{bmatrix}\Delta t_{c2} + \Delta t_{c3} + \Delta t_{c6}\\ 2\cdot\Delta t_t + \Delta t_{c4}\\ 2\cdot\Delta t_t + \Delta t_{c5}\end{bmatrix} = \begin{bmatrix}0\\2\\2\end{bmatrix}\cdot\Delta t_t + \begin{bmatrix}2+4\\2+2\\2+2\end{bmatrix}\cdot\begin{bmatrix}\mathrm{p}_{unit1}\\ \mathrm{p}_{unit2}\\ \mathrm{p}_{unit3}\end{bmatrix}$$

To illustrate the method: all instructions in the code are unrolled and the method creates different gamma-graph nodes and corresponding transfers, as visible in Figure 6 for a call of fib(3). Figure 75 shows the forming of compute and transfer segments. From the computation and communication segments the corresponding computing times Δt_{ti} and communication times Δt_{ti} can be distinguished. The computing time can be seen, e.g. as a function of number of instructions n_{instr} and computing unit performance p_{unit} $\Delta t_{ci} = f(n_{instr}, p_{unit})$. Each chain of computing segments is associated to one computing unit. For three units $n_{instr} = 3$. Figure 77 shows a time model structure for compute and transfer segments. A simple set of linear equation can be built to generate the runtime Δt_{end} with $\Delta t_s = 0$. The latency times for the units are:

The runtime is the maximal runtime of each of the units and is a function of the hardware-transfer time Δt_t and the performance of the units $\vec{p}_{unit}$:

The idle-times are on:

$$\Delta t_{end,nunits=3}(\Delta t_t, \vec{\mathrm{p}}_{unit}) = \max\begin{pmatrix}\Delta t_{end,u1}\\ \Delta t_{end,u2}\\ \Delta t_{end,u3}\end{pmatrix}$$

- unit 1 or 2: $\Delta t_{idl1} = [\Delta t_{c3}] - [2\cdot\Delta t_t + \Delta t_{c4}]$
- unit 1 or 2: $\Delta t_{idl2} = [\Delta t_{idl1}(\Delta t_t) + \Delta t_{c6}] - [2\cdot\Delta t_t + \Delta t_{c5}]$

For two units: $n_{units} = 2$. Figure 78 shows an option for 2 units, marked as combination (c1) and combination (c2). The latency times for the units are either for combination 1:

$$\begin{matrix}\Delta t_{end,u1,c1} \\ \Delta t_{end,u2,c1}\end{matrix} = \begin{bmatrix}\Delta t_{c1} + \Delta t_{c3} + \Delta t_{c6} \\ 2 \cdot \Delta t_t + \Delta t_{c4}\end{bmatrix} = \begin{bmatrix}0 \\ 2\end{bmatrix} \cdot \Delta t_t + \begin{bmatrix}5+6+8 \\ 2\end{bmatrix} \cdot \vec{p}_{unit}$$

or for combination 2:

$$\begin{matrix}\Delta t_{end,u1,c2} \\ \Delta t_{end,u2,c2}\end{matrix} = \Delta t_{c1} + \Delta t_{c7} + \begin{bmatrix}\Delta t_{c1} + \Delta t_{c2} + \Delta t_{c7} \\ 2 \cdot \Delta t_t + \Delta t_{c5}\end{bmatrix} = \begin{bmatrix}0 \\ 2\end{bmatrix} \cdot \Delta t_t + \begin{bmatrix}1+6+1 \\ 2\end{bmatrix} \cdot \vec{p}_{unit}$$

The runtime is the maximal runtime of each of the units of both combinations:

$$\Delta t_{end,nunits=2}(\Delta t_t, \vec{p}_{unit}) = \max\left(\max\begin{pmatrix}\Delta t_{end,u1,c1} \\ \Delta t_{end,u2,c1}\end{pmatrix}, \max\begin{pmatrix}\Delta t_{end,u1,c2} \\ \Delta t_{end,u2,c2}\end{pmatrix}\right)$$

For one unit: $n_{unit}=1$, which is illustrated in figure 79. The latency time for one unit is $\Delta t_{end,u1} = [0] \cdot \Delta t_t + [10] \cdot p_{unit}$.

For the overall optimization: In this form Δt_{end} is a function of Δt_{ci}, Δt_{ti} and n_{units}. This can be expressed directly by $\Delta t_{end} = f(\Delta t_{ci}, \Delta t_{ti}, n_{units})$. So the optimization is for a given set of hardware with Δt_t, $\vec{p}_{unit}$ is:

$$\Delta t_{end,optimal}(\Delta t_t, \vec{p}_{unit}) = \min\begin{Bmatrix}\Delta t_{end,nunits=3} \\ \Delta t_{end,nunits=2,c1} \\ \Delta t_{end,nunits=2,c2} \\ \Delta t_{end,u1}\end{Bmatrix}$$

This illustrates the effect of the new technical optimization approach differing completely from the known prior art SOTA approach (cf. V. Sarkar *"Fundamentals of parallel programming module 1 parallelism"*, p. 58ff).

Optimization approach b): *Building gamma-graphs following task parallelism*

According to the disclosure of the present invention, the result of the system can be illustrated as a tensor or a graph, depending on the form of representation, see Figure 80. If all instructions in the code are unrolled, the method creates different gamma-graph nodes and corresponding transfers, as visible in figure 81 for a call of fib(3).

Now, for each node in the gamma-graph, the system 1 computes whether the overhead of running a child-node in parallel by paying the prize of "*transferring the data*" (this can also be seen as time of a context switch) is beneficial.

- serial effort: $t_{serial\Gamma} = \sum \Delta t_{instr}$
- parallel effort: $t_{parallel\Gamma 2} = t_{serial\Gamma parent} - t_{serial\Gamma child} + t_{contextsw}$
- time overhead: $\Delta r_a(t_{contextsw}) = \frac{t_{serial\Gamma parent}}{t_{parallel\Gamma parent}}$

Following the Fibonacci example, this leads to the computation graph with corresponding t_{serial} and $t_{parallel}$ times for each node as indicated in figure 82. In a compiler, this can be seen as given the compiler a transfer time (TT) for a platform and the compiler is capable of forming a TT-agnostic computation graph. For example by computing a computation graph for the Fibonacci code fib(5) with 3 different transfer times = 1, 5, 10, the method is capable of forming the different computation graphs for corresponding platform (for n=5 and TT = 1, 5, 10 are so low to be able to easier show the effect graphically). Figure 83 shows exemplary three different computation graphs for TT = 1, 3, 10. For a low TT = 1 it is beneficial to compute all gamma-nodes in parallel. For a higher value of TT, e.g. TT = 5, gamma node 4 and 5 are in serial with gamma-node 2 and 5, 6, 8 with 3, but these two can be run in parallel. These results correspond well with results e.g. shown in V. Sarkar "*Fundamentals of parallel programming module 1 parallelism*".

(v) IC-design using the inventive system 1

The inventive system 1 can be applied (see figure 106/110) for both designing and optimizing VLSI designs. Circuit design is about arranging transistors to

perform a particular logic function. From the design the delay and power can be estimated. Every circuit can be represented as a schematic or in textual form as a netlist. Simply illustrated (see Fig. 111a), digital logic can be divided into combinational circuits (Boolean logic), whose outputs depend only on the present inputs (series of logic gates), and sequential circuits, whose building blocks are registers (flip-flops) and latches.

The Inventive System 1 can be used to determine the computation block nodes. The computation block nodes in turn can be used to determine the digital logic, which will be illustrated by the example of the 2D heat equation. Fig. 67 shows that a computation block node for this algorithm consists of 5 reads, 1 compute and 1 write (Fig. 111b). The result of a simple circuit is shown in Fig. 111c. The reads become the sequential circuits consisting of 5 flip-flops (together they form the register, though, the effective register size is determined by the bit length of the individual data). A rising-edge of the clock (time k) can be assumed, where for the required hold-time is waited to be sure that the correct values (logical 0 or 1) for u[k][i+1][j], u[k][i-1][j], u[k][i][j+1], u[k][i][j-1] and u[k][i][j] are present at the output of the flip-flops. This is basically our read. The data can now propagate through the combinational circuit (computational block), which results from the arithmetic consisting of the required adder, shifter (multiplication by 4), subtractor (inverter and adder combined) and multiplier. At the output of the computational block the value u[k+1] arises (equivalent to the write operation) after the setup time, which is the amount of time required for the input of a flip-flop to be stable before a next clock edge.

The register on the right side is nothing else than the value at the next time k+1, which is needed for the next iteration on the left side. From this results that the values can be written directly into the same register. This thought experiment can now be carried out for each point on the computation matrix. Already for two points on the matrix (Fig. 111d) it becomes obvious that data are now written crosswise or that only one register per matrix point is needed. The computational block (CB) always consists of the same combinational circuit, although in the electronic implementation the propagation delays will never be the same. It is therefore important to keep an eye on the critical paths that limit the operating speed of the system and require attention to timing details.

Real world settings are always limited in space with VLSI design. An infinite parallizability is not possible, so certain (not all) computational blocks are combined and processed sequentially. To make this possible for a finite number of registers, a multiplexer is connected between the output of the registers and the input to the CB. A multiplexer chooses the output from among several inputs based on a select signal. At the same time, a demultiplexer is connected between the output of the CB and the register input (see Fig. 111e) to feed the data back correctly, whereby the design optimization process might show that the demultiplexer is not always need. What is processed per clock rate on the ideal, parallel circuit is now processed in several clock cycles in the more realistic circuit (Fig. 111f/g) – as much parallel as possible in as many cycles until a complete iteration is completed. Using the 4x5 matrix from Fig. 67 as an example, this would mean: With 6 parallel CBs: 1 cycle to calculate k+1, with 3 parallel CBS: 2 cycles (2x3 parallel) including a higher delay do to multiplexing.

VLSI designers always must make trade-offs between area, throughput, latency, power consumption and energy of executing a task. The optimal circuit is always located somewhere on an inverse curve (see Fig. 111h). If There is a lot of area available, parallelization can be done well. If there is less area available, the system must multiplex, which leads to a higher delay. Somewhere in between lies the optimal point between area and number of parallel CBs. Since the Inventive System 1 can give the optimal code for a given unit, it can also be used to find the optimal circuit for a given area, helping in the iterative design process refined as actual module sizes and critical paths become known.

There is to say that the exact implementation of the parallelization on an FPGA or of an ASICs would depend on the specifics of the problem and the design requirements. It may require careful consideration of the memory and especially the communication requirements between systems, as well as the previous stated trade-offs between area, throughput, latency, power consumption, and energy that are common in VLSI design.

Hence, the Inventive System 1 employs a set of heuristics and algorithms to identify and manipulate patterns in the design space to generate novel and non-obvious solutions. The potential design solutions are evaluated based on a set of predetermined metrics, such as power consumption, area, and performance, and the optimal design solution is selected. The selected design solution is further optimized

through a series of iterative steps, using the Inventive System 1 to explore and manipulate the design space.

In the method for optimizing the parallel VLSI design process, registers and clocks play a crucial role in ensuring the proper functioning of the circuit. The Inventive System 1 considers the read and write operations of the registers, as well as the clock timing constraints, to generate a design that meets the desired performance and functionality requirements.

Furthermore, the Inventive System 1 considers the propagation delay through the arithmetic logic to optimize the timing of the circuit. This includes considering the delay in each logic gate and the routing between the gates to minimize the overall delay of the circuit. The Inventive System 1 employs a variety of techniques, such as pipelining and parallelism, to minimize the propagation delay and maximize the performance of the circuit.

However, while optimizing the design for performance, the Inventive System 1 also considers the area constraints of the target implementation platform, such as FPGA cells. The method ensures that the resulting design fits within the available resources of the implementation platform, while also meeting the desired performance and functionality requirements.

In summary, the method for optimizing the parallel VLSI design process using Inventive System 1 considers the read and write operations of registers, clock timing constraints, propagation delay through the arithmetic logic, and area limitations of the implementation platform. This results in a highly efficient and scalable VLSI design that meets the desired performance and functionality requirements, while also being feasible for implementation in the target platform.

Inventive system 1 and inventive method in the context of other prior art systems

It is to be noted that the various approaches, known the prior art, differ essentially from the present invention by their technical approach. In the following, the

essential difference to three of the prior art systems are detailed explained in order to further illustrate the inventive system and method: (a) *Slicing based code parallelization for minimizing inter-process communication* by M. Kandemir et al. (hereafter Kandemir) discloses a method for scalable parallelization that minimizes inter-processor communication distributed memory multi-core architectures. Using the concept of iteration space slicing, Kandemir discloses a code parallelization scheme for data-intensive applications. This scheme targets distributed memory multi-core architectures, and formulates the problem of data-computation distribution (partitioning) across parallel processors using slicing such that, starting with the partitioning of the output arrays, it iteratively determines the partitions of other arrays as well as iteration spaces of the loop nests in the application code. The goal is to minimize inter-processor data communications based on this iteration space slicing based formulation of the problem. However, Kandemir does this with the portioning of output array, by iteratively determining the partition of other arrays in the application code (see p. 87). This is a different approach, because the inventive method, disclosed herein, does not include an iterative determination of array parts, which are combined. The inventive method gets this information directly from the code. As disclosed on p. 88 Kandemir, program slicing was originally introduced by Weiser in a seminal paper. Slicing means extracting statements form a program that potentially have an influence on a specific statement of interest, which is the slicing criterion (cf. *Advanced slicing of sequential and concurrent programs* by J. Krinke). Slicing techniques showing similar effects as the well-known points data/control dependences (cf. *Computer architecture* by D. Patterson, p. 150ff) and data flow analysis (e.g. [8]). Our method resembles on the first impression these methods. This comes from the fact, that statements, blocks in programs, data dependencies are one of the most central points in programing and compiling, but the inventive method, disclosed herein, has a different perspective. The novelty is in this new approach providing a new perspective, respectively the definition of the computation block nodes, as indicated by the report. The inventive method explicitly takes each variable as a unique information (represented by one particular bit-pattern) and as the computing instruction (= statement) can only be performed with available bit-patterns (in accessible storages, e.g. in CPU registers), the inventive method extracts the time between the modification and the use of a variable at a "location" (e.g. a register). No SOTA compiler known extracts this time out of the (source) code nor focus on the single data entities (bit-pattern) in a statement – all known prior art systems use always as a focus the result of a complete statement. As Kandemir discloses (p. 89), using iteration space slicing, one can answer questions such as "*Which iterations of*

which statements might affect the values of a given set of elements from arrayA". This shows a different perspective, where the inventive method derives exactly this affection in a generic form of finding the «read» and «write» on each element of the array and brings this delay time of "reading" and "writing" in correlation to the time needed to transfer an element. It finds this »affect» and do not try to find it by «simulating» the interaction with linear algebra methods, respectively by iterative methods. On p. 91, Kandemir summarizes the function that returns the set of loop iterations to be assigned to processor p from loop nest s, where Zp,r is the set of data elements accessed by processor p from array Ar. In contrast to this, the inventive system and method is not based on this approach: the inventive method and system derives first all dependencies of the code (also in the nested loops), then it creates matrices for all code in the program and at the end the system maps/optimizes by combining of the matrices all operations of a program. Kandemir's method creates a matrix for the array loop index dependencies and derives by assigning them to processors and then finds (see Kandemir p. 92) how very iterative this approach is needed to be by taking the Presbuerger set and then generating a code as output "(a series of potentially nested loops)" and then iterates over the unknowns. Kandemir's approach does not take hardware specification into account either; (b) *Automatic Parallelization: Executing Sequential Programs on a Task-Based Parallel Runtime* by A. Fonseca (hereafter Fonseca) discloses another system of the prior art for automatically parallelizing sequential code in modern multicore architectures. Fonseca discloses a parallelizing compiler that analyses the read and write instructions, and control-flow modifications in programs to identify a set of dependencies between the instructions in the program. Afterwards, the compiler, based on the generated dependencies graph, rewrites, and organizes the program in a task-oriented structure. Parallel tasks are composed by instructions that cannot be executed in parallel. A work-stealing-based parallel runtime is responsible for scheduling and managing the granularity of the generated tasks. Furthermore, a compile-time granularity control mechanism also avoids creating unnecessary data-structures. Fonseca focuses on the Java language, but the techniques could possibly be applied to other programming languages. However, in contrast to the inventive method, discloses in this application, in Fonseca's approach, in order to automatically parallelize the program, it is necessary to analyze the memory accessed to understand dependencies between parts of the program (cf. p. 6, Fonseca). The inventive method, disclosed herein, is different: Fonseca uses data-groups and memory layouts and then it checks the dependencies. This is not the same scope as the inventive method, since as Fonseca explicitly holds that it is task

parallelism. For instance in the Fibonacci example, as discussed above, the cost of creating a new task is higher than the cost of executing the method for a low input number (see Fonseca p. 7). This shows that this is not the same approach, as the inventive method handles this example completely different. On the other side, this also directly the prove for a technical problem which can be solved by the inventive method. Further, Fonseca (see p. 9) has to define the main requirements for the position of the future creation. In contrast, the inventive method knows where to place each instruction, respectively statement, depending on the data dependency of the single information / variable. Fonseca (see p. 9) also discloses that an algorithm 18 has to be used to find the best position to create the future. The inventive method, in contrary, exactly places the instructions to the position based on the new scope. In Fonseca (see p. 10), the used operations that are commutative and associative. The inventive method does not base its approach on this form, because it excludes e.g. division (used in a lot of mathematical models). This limitation is also typically for map-reduce approaches (as discussed herein); (c) The publication US 2008/0263530 A1 discloses a system for converting application code into optimized application code or into execution code suitable for execution on an architecture comprising at least a first and a second level of data memory units. The method obtains application code, the application code comprising data transfer operations between the levels of memory units. The method further comprises converting at least a part of the application code. The converting of application code comprises scheduling of data transfer operations from a first level of memory units to a second level of memory units such that accesses of data accessed multiple times are brought closer together in time than in the original code. The converting of application code further comprises, after the scheduling of the data transfer operations, deciding on layout of the data in the second level of memory units to improve the data layout locality such that data which is accessed closer together in time is also brought closer together in the layout than in the original code. US 2008/0263530A1 allows improving layout locality (see US 2008/0263530A1 p. 4, para. 0078). In contrast, the inventive method, disclosed herein, has a different scope, since the inventive method finds this form of "locality" and orders the instructions in a way that instructions are grouped, which must be "local" - then this inherent form of the code is placed in matrices and then there can be a generic, not-iterative, respectively by combining elements and then set the optimal mapping to a given hardware, which leads always in a concurrent form of the code. Iterative solution methods can miss solutions and end with a non-clear result. Further, US 2008/0263530A1 (p. 6, para. 0092) discloses that it can be regarded as a complex, non-linear problem, for which a

reasonable, near-optimal, and scalable solution is provided. In contrast, the inventive method is not a non-linear problem, rather it prevents to get a complex, non-linear problem, which needs an iterative, numerical solution / optimization approach / algorithm. US 2008/0263530A1 (p. 6, para. 0093) holds that its access locality is improved by computing the reuse vectors and applying them to find an appropriate transformation matrix T. In contrast, the inventive method, disclosed herein, does not need a transformation matrix, nor computing of such one. It reads the inherent logic connection between the array operations based on the given code (in form of loop-definition or loop blocks in compiler IR languages or jump definitions in assembler code). Finally, US 2008/0263530A1 (p. 6, para. 0093) holds that having fixed T, thereafter the placement M for the arrays accessed in the loop-nest is fixed, whose placement is not already fixed. This discloses an iterative and numerical solver based approach. The inventive method, disclosed herein, reads this dependencies in the code, without an iterative solving technique using linear algebraic methods to finding solution of systems of equations.

List of reference signs

1 Auto-parallel-compiler system
11 Lexer/Parser
12 Analyzer
13 Scheduler
14 Computational block chains module
15 Matrix builder
151 Computation matrices
152 Transfer matrices
153 Task matrices
16 Optimizer module
17 Code generator
2 Parallel processing system/Multi-processer system
21 Processor units
210 Central Processing Units (CPU)
2101 Control Unit
2102 Processor (Single-core micro-controller CPU)
21021 Register
21022 Combinational logic
2103 Core/Processor (Multi-core micro-controller CPU)
211 Graphics Processing Units (GPU)
212 Sound chips
213 Vision Processing Unit (VPU)
214 Tensor Processing Units (TPU)
215 Neural Processing Units (NPU)
216 Physics Processing Units (PPU)
217 Digital Signal Processors (DSP)
218 Synergistic Processing Units (SPU)
219 Field-Programmable Gate Arrays (FPGA)
22 Memory units
221 Primary storage units
2211 Processor registers
2212 Processor caches
22121 L1-cache
22122 L2-cache

……..

2212x Lx-cache

2213 Random Access Memory (RAM) units

222 Secondary storage units

2221 Hard Disk Drives (HDD)

2222 Solid-State Drives (SSD)

2223 Universal Serial Bus (USB) memory

2224 Flash drives

2225 Optical storage devices (CD or DVD drives)

2226 Floppy disk drives (FDD)

2227 RAM disks

2228 Magnetic tapes

223 Tertiary storage units (tape backup etc.)

23 Memory bus

231 Address bus

232 Data bus

24 Memory Management Unit (MMU)

25 Input/Output (I/O) interfaces

251 Memory-mapped I/O (MMIO) or port-mapped I/O (PMIO) interfaces

252 Input/Output (I/O) channels (processors)

3 Program code

31 Serial source code

311 High level language

3111 C/C++

3112 phyton

3113 Java

3114 Fortran

3115 OpenCL (Open Computing Language)

3112 Parallel programming language

31121 Apache Beam

31122 Apache Flink

31124 Apache Hadoop

31125 Apache Spark

31126 CUDA

31127 OpenCL

31128 OpenHMPP

31129 OpenMP for C, C++, and Fortran (shared memory and attached GPUs)

3113 Low-level language (machine code/assembly language)

32 Auto-parallelized symmetric target code

321 Low-level language (assembly language)

322 Machine language (computer's instruction set (code directly executed by central processing units))

322 Basic set of elementary instructions

3221 Arithmetic operation instructions

3222 Logical operation instructions

3223 Variable and array declarations operations

3224 Compare operation instructions

3225 Code flow instructions/Memory operation/I/O operation

33 Nodes

331 Data nodes (holding certain data values)

3311 Input data node for READ ($Datanode_{in}$ (read access))

3312 Output data node for WRITE ($Datanode_{out}$, write access))

332 Operation nodes (performing an operation)

333 Computation block nodes (CB1, CB2,…,CBx)

334 Control flow nodes

3341Branch nodes

3342 Hidden branch nodes

3343 Loop branch nodes

335 Condition nodes

336 Gamma nodes (tasks)

34 Chains

341 Computational chains (chains of computation block nodes)

342 Operational chains (chains of operation nodes)

35 Latency time Δt (time to transfer data from one process to another)

351 $\Delta t_{readwrite}$ = time between a write and a read access to a data node

352 $\Delta t_{computation}$ = time to compute all operation nodes in a computation block node

353 Total (aggregated) latency time Δt_{total}

36 Tasks

36i Number of computation block nodes in a task i

4 Network

41 Network controller

Claims

1. Symmetric auto-compiler system (1) for hardware-optimized auto-parallelization of a program code (3) for execution by a multi-core or multi-processor parallel processing system (2) having a plurality of processing units (21) simultaneously processing instructions on data in the parallel processing system (2) by executing the program code (3), the auto-compiler system (1) comprising means for transforming a serial source code (31) of the program code (3) written in a programming language into a parallel processing machine code (32) comprising a number of instructions executable by the plurality of processing units (21) of the parallel processing system (2) or controlling the operation of the plurality of processing units (21),

wherein the parallel processing system (2) comprise memory units comprising at least main execution memory units (221/2212) comprising a plurality of memory banks for holding data of at least parts of the processing code (32), and transition buffer units (221/2211) comprising high speed memory for storing starting locations of the processing code (32) and data segments at least comprising branch or jump instructions and/or used memory references and data values, wherein the main execution memory units (2212) provide a slower access time than the transition buffer units (2211),

wherein the execution of the processing code (32) by the parallel processing system (2) comprises the occurrence of latency times (26), the latency times being given by idle time of a processing unit (21) between transmitting data back after having processed a specific block of instructions of the processing code (32) on the data by the processing unit (21)and receiving data necessary for execution of a consecutive block of instructions of the processing code (32) by said processing units (21), characterized,

in that the compiler system (1) comprises a parser module (11) for translating the serial source code (31) into a code (32) with a flow of elementary instructions executable by the processing units (21), the elementary instructions selectable out of a limited, processing-unit-specific set of elementary instructions and the elementary instructions comprising exclusively elementary arithmetic and logic operations (321/322) and/or elementary control and memory operations (325) for the number of processing units (21),

in that the parser module (11) comprises means for partitioning the code (32) of said elementary instructions into computation block nodes (333), each computation block node (333) consisting of a smallest possible segmentation of a non-further decomposable sequence of elementary instructions of the code (32) processable by a single processing unit (21), the smallest possible segmentation of the elementary instructions being characterized by a sequence of elementary instructions framed by consecutive read and write instructions, said sequence being not further decomposable by smaller sequences of elementary instructions between consecutive read and write instructions, and the read and write instructions needed to receive data required for processing said sequence of elementary instructions by the processing unit (21) and transmit back data after processing by the sequence;

in that the compiler system (1) comprises a matrix builder (15) for generating matrices (151,...,153) out of computation chains (34) portioned from the code (32), the matrices (151,...,153) comprising computation matrices and transfer matrices (151/152) and task matrices (153), wherein each column within a computation matrix (151) comprises computation block nodes (333) being simultaneously processable based on executability of read and write instructions transferring data required for processing of the computation block nodes (333), wherein the transfer matrix contains to each computation block nodes (333) transfer and process properties at least characterizing data transfer properties from one to a consecutive computation block node (333) at least comprising data size of transmitted data and identification of a source computation block node (333) and target computation block node (333) of the data transfer and/or process characteristics on one of the plurality of processing units (21),

in that tasks (56) of a task matrix (153) are formed by the matrix builder (15) wherein in case of computation block nodes (333) each having a different read associated, the tasks (56) are formed by splitting the computation block nodes (333) of a column of the computation matrix (151) evenly to the number of the plurality of symmetric processing units (21) forming for each of the plurality of processing unit (21) per column of the computation matrix (151) one task (56), and splitting remaining computation block nodes (333) based on a predefined scheme to at least a part of said tasks (56), wherein in case of computation block nodes (333) at least partially having a read with identical data transferred, the tasks (56) are formed by minimizing the number of reads uniformly or essentially uniformly over the number of processing

units (21) and/or if a predefined offset value is exceeded by minimizing an integrated processing time uniformly over each of the processing units (21),

in that the compiler system (1) comprises an optimizer module (16) using a matrix optimization technique minimizing an aggregated occurring latency time (26) integrating all occurring latency times (261) by providing an optimized structure of the tasks (56) within the task matrix (153), wherein each column of the task matrix (153) forms a computation chain (34) by one or more tasks (333) creating an ordered flow of computation block nodes (333) to be executed by one of the plurality of processing units (21),

in that the compiler system (1) comprises a code generator (17) for generating the parallel processing machine code (32) for the plurality of processing units (21) with an optimized aggregated latency time (26) based on the computation chains (34) given by the optimized task matrix (153).

2. Compiler system according to claim 1, characterized in that parallel processing system(2) is a symmetric multi-core system and/or a multi-processor system having the plurality of processing units (21), wherein the symmetric multi-core system and/or multi-processor system is a system (1) which has multiple cores (2103) or processors (2102) that are all exactly or essentially exactly the same.

3. Compiler system according to claim 2, characterized in that the processors (2102) and/or cores (2103) have the same storage architecture and technical processor characteristics.

4. Compiler system according to claim 3, characterized in that the technical characteristics at least comprise having the same processor (2102) or core (2103) performance with equal or essentially equal numbers of instructions per second processed.

5. Compiler system according to one of the claims 1 to 4, characterized in that the predefined scheme comprise splitting the remaining computation block nodes (333) by adding additional transfers for processing a respective computation block nodes (333).

6. Compiler system according to claim 5, characterized in that the defined scheme for splitting the remaining by adding additional transfers comprises distributing the remaining computation block nodes (333) to the plurality of processing units (21) evenly by adding transfers, as long as the added latency times by the additional transfers is smaller than the corresponding distributed computation times and (n_{evenCBs}, $n_{\text{partialCBs}}$) = divmod($n_{||}$, n_{units}) is true.

7. Compiler system according to one of the claims 1 to 4, characterized in that the predefined scheme comprise splitting the remaining computation block nodes (333) by adding to each task (56) one computation block node (333) until no remaining computation block nodes (333) are left.

8. Compiler system according to claim 1, characterized in that parallel processing system(2) is an asymmetric multi-core system and/or a multi-processor system having the plurality of processing units (21), wherein the asymmetric multi-core system and/or multi-processor system is a system (1) which has multiple cores (2103) or processors (2102) where at least some of the multiple cores (2103) or processors (2102) differ in their processor (2102) or core (2103) performance different number of instructions per second processed.

9. Compiler system according to claim 8, characterized in that the computation block nodes (333) of a column of the computation matrix (151) are split evenly to the number of the plurality of processing unit (21) by adapting the length of the tasks (56) assigned the different processing units (21) in respect to the performance of the particular processor (2102) or core (2103) based on processed instructions per second.

10. Compiler system according to one of the claims 1 to 9, characterized in that the source programming language is a high-level programming language.

11. Compiler system according to claim 10, characterized in that the high-level programming language comprises C and/or C++ and/or python and/or java.

12. Compiler system according to one of the claims 1 to 11, characterized in that the transition buffer units comprise cache memory modules and/or L1-cache.

13. Compiler system according to one of the claims 1 to 12, characterized in that the latency time comprises register access time and/or L1-cache access time and/or memory access time and/or I/O operation time and/or data network transfer time and/or processor configuration time.

14. Compiler system according to one of the claims 1 to 13, characterized in that at least two of the processing units have different sets of elementary instructions.

15. Compiler system according to one of the claims 1 to 14, characterized in that the arithmetic operations comprise the operations addition, subtraction, multiplication, and division.

16. Compiler system according to one of the claims 1 to 15, characterized in that the logic operations comprise a number of logical expressions like equal, not equal, greater than, less than, greater than or equal, less than or equal.

17. Compiler system according to one of the claims 1 to 16, characterized in that the control operations comprise a "branch expression" and/or a "loop expression".

18. Compiler system according to one of the claims 1 to 17, characterized in that the computation block nodes (333) are generated by creating a mapping of read and write of data nodes (331) by their operation nodes (332) by mapping the interactions of the read and write of the single instructions.

19. Compiler system according to one of the claims 1 to 18, characterized in that the computation block nodes (333) consist of connected operation nodes (332), the operation nodes (332) being composed in one chain (34) having one or more input data nodes (331/3311) and one or more output data nodes (331/3312).

20. Compiler system according to claim 19, characterized in that the computation block nodes (333) are connected to next computation block nodes (333) or to control flow nodes (334) or to one or more input data nodes (331/3311) and one or more output data nodes (331/3312) building a chain (34).

21. Compiler system according to one of the claims 1 to 20, characterized in that each pair of computation and transfer matrices is realized forming a single numerical matrix and/or multi-dimensional numerical tensor, respectively.

22. Compiler system according to claim 21, characterized in that the numerical matrix optimization is realized as tensor optimization.

23. Compiler system according to one of the claims 21 or 22, characterized in that the numerical matrix optimization or tensor optimization by means of the optimizer module is based on a dedicated applied machine learning structure.

24. Compiler system according to one of the claims 1 to 23, characterized in that each computation block node (333) of a computation chain (34) is numbered along the ordered flow of computation block nodes (333) within the computation chain (34) by assigning block numbers to the computation block nodes (333), wherein in the computation and transfer matrix, the block numbers of the computation block nodes (333) are equal to the column of the matrices, and wherein the computation block nodes (333) with the same block number are distributed to different rows of the same column.

25. Compiler system according to one of the claims 1 to 24, characterized in that in the computation matrix, each row contains a computation chain (34) of a processing unit (21), wherein the cells of the computation matrix comprise the computation block nodes (333) of the computation chain (34), and wherein in the transfer matrix, the cells contain the required transfer properties to and/or from other computation block nodes (333).

26. Compiler system according to claim 25, characterized in that the transfer matrix with the cells having the transfer properties comprise at least of what information is needed in another computation block node (333).