A Data-Dependent Parallel Computation Method Based on LLVM Intermediate Representation (Postprint)

Authors: Zhu Yan, Zhong Lujie

Date: 2018-12-13T00:00:00+00:00

Abstract

Low Level Virtual Machine (LLVM) is a widely-used compiler framework whose Intermediate Representation (IR) contains rich program analysis information; numerous LLVM-based works are developed based on IR. Data dependency relationships have important applications in error detection, localization, and program debugging. IR-based data dependency computation predominantly employs serial iterative approaches, but scalability is suboptimal when handling large-scale IR files. To address this, we explore the parallelism potential in instruction reads and writes during data dependency computation, and leveraging the parallel computing advantages of Graphics Processing Units (GPUs), propose DRPC, a parallel computation method for data dependency relationships based on LLVM IR. This method takes IR as input and achieves efficient computation of program data dependency relationships through a CPU-GPU collaborative approach. Experimental results demonstrate that for the SPEC benchmark suite, DRPC obtains maximum speedups of 3.48x and 4.91x in direct and transitive data dependency relationship computation, respectively.

Full Text

Preamble

A Parallel Data Dependence Computation Method Based on LLVM Intermediate Representation

Zhu Yan, Zhong Lujie
(College of Information Engineering, Capital Normal University, Beijing 100048, China)

Abstract: The Low Level Virtual Machine (LLVM) is a widely-used compiler framework whose intermediate representation (IR) contains abundant program

analysis information, serving as the foundation for numerous LLVM-based research efforts. Data dependence relationships play a crucial role in error detection, fault localization, and program debugging. While IR-based data dependence computation typically employs serial iterative approaches, such methods exhibit poor scalability when handling large-scale IR files. To address this limitation, we explore the parallelism potential in instruction read/write operations during data dependence calculation and propose DRPC, a parallel data dependence computation method based on LLVM IR that leverages GPU parallel computing advantages. Our approach takes IR as input and achieves efficient program data dependence computation through CPU-GPU collaborative processing. Experimental results on the SPEC benchmark suite demonstrate that DRPC achieves maximum speedups of 3.48× and 4.91× for direct and transitive data dependence computation, respectively.

Keywords: LLVM; intermediate representation; data dependence; graphics processing unit; scalability

0 Introduction

LLVM (Low Level Virtual Machine) is an open-source compiler framework that provides a Static Single Assignment (SSA)-based intermediate representation (IR), which serves as the foundation for subsequent optimizations and related work. IR contains rich program analysis information, and data dependence analysis represents an important program analysis technique with widespread applications in program optimization, maintenance, and debugging. For instance, instruction scheduling guided by inter-instruction dependencies and more precise fault localization based on dependencies between program entities both rely on accurate data dependence information. LLVM IR-based data dependence analysis typically employs serial iterative methods, which exhibit insufficient scalability when applied to large-scale programs.

1.1 Data Dependence

Data dependence is a typical dependence analysis method that primarily reflects the read-write dependency relationships that must be observed between program statements. Based on whether multiple consecutive data dependencies exist between statements, data dependence can be categorized into direct data dependence and transitive data dependence. Related definitions are as follows:

Definition 1: Given a sequence of n sequentially executed statements S_1, S_2, ⋯, S (n ≥ 2), where S_1 is called the start statement, S the end statement, and the rest intermediate statements. The dependence sequence D = (D_1, D_2, ⋯, D $_1$) formed by dependence relationships D (1 ≤ i ≤ n−1) between adjacent statements is called a dependence chain. Let read(S) denote the set of memory read variables for statement S , and write(S) denote the set of memory write

variables for statement S .

Direct Data Dependence (dependence chain length of 2, i.e., n = 2) is defined as:

Definition 2 (Flow Dependence, FD): If write(S) read(S $_1$) ≠ for i = 1, 2, ···, n−1, then flow dependence exists between S and S $_1$, denoted as S FD S $_1$.

Definition 3 (Anti-dependence, AD): If read(S) write(S $_1$) ≠ for i = 1, 2, ···, n−1, then anti-dependence exists between S and S $_1$, denoted as S AD S $_1$.

Definition 4 (Output Dependence, OD): If write(S) write(S $_1$) ≠ for i = 1, 2, ···, n−1, then output dependence exists between S and S $_1$, denoted as S OD S $_1$.

When n > 2, if the start and end statements have multiple consecutive data dependencies with transitive characteristics, this relationship is called transitive dependence.

Definition 5 (Transitive Dependence, TD): For a statement sequence S , S $_1$, S $_2$, ···, S , if S and S satisfy the condition that write(S) read(S $_1$) ≠ for all m where i ≤ m ≤ j−1, then transitive dependence exists between S and S , denoted as S TD S . Notably, since adjacent statements in such a sequence all exhibit flow dependence, the sequence forms a flow dependence chain.

Consider the program fragment shown in Figure 1: see original paper. Since write(S_2) = {a}, read(S_3) = {a}, write(S_3) = {b}, read(S_5) = {b}, write(S_5) = {d}, we have write(S_2) read(S_3) ≠ and write(S_3) read(S_5) ≠ , establishing transitive dependence between S_2 and S_5.

1.2 LLVM IR

[Figure 2: see original paper] illustrates the LLVM compilation framework, which primarily consists of three components: frontend, optimizer, and backend. LLVM' s custom intermediate representation IR is a crucial component of this framework. IR serves not only as the basis for LLVM optimization and backend-related work but also as the main input for various LLVM-based program analysis applications. IR generally has three forms: readable assembly format, binary code format, and in-memory intermediate representation.

LLVM IR text typically provides program analysis information including function declarations, basic blocks, variable declarations, and function calls in instruction set form. Typical LLVM instruction categories include memory access instructions, terminal instructions, binary instructions, bitwise binary instructions, and other instructions. Figure 1: see original paper shows the IR text content corresponding to the code fragment in Figure 1: see original paper.

1.3 LLVM IR-Based Data Dependence Computation

In LLVM IR, memory read/write instructions relevant to data dependence computation include alloca, store, and load. The alloca instruction represents variable declaration, store represents memory write operations, and load represents memory read operations. Related variables in these instructions are marked with the "%" symbol. By tracing the characteristics of relevant read/write instructions, we can extract the program analysis information required for data dependence computation. For example, by tracing store and load instructions in IR and performing analysis, we can compute the Write and Read sets needed for data dependence calculation and subsequently complete the dependence computation.

Using the program fragment in Figure 1: see original paper as an example, we identify memory read/write instructions related to variables a, b, and d in the IR. Let Sstore_1 denote the store instruction marked in Figure 1: see original paper, and Sload_1 denote the load instruction marked . We have read(Sstore_1) = {1}, write(Sstore_1) = {%a}, read(Sload_1) = {%a}, write(Sload_1) = {%0}. Since write(Sstore_1) read(Sload_1) $\neq$, we can compute the flow dependence between Sload_1 and Sstore_1. The characteristic trace path is shown by the dashed arc with an arrow in Figure 1: see original paper.

1.4 Parallelism Analysis in IR-Based Data Dependence Computation

Parallelism in Direct Data Dependence Computation. Direct data dependence computation requires considering the read-write intersection between start and end instructions. On one hand, we can partition computation tasks using the non-consecutive read-write intersection characteristics between different start and end instructions. On the other hand, we can further exploit memory access characteristics during instruction information reading and data dependence relationship updates to achieve data parallelism.

Parallelism in Transitive Data Dependence Computation. Although transitive data dependence computation exhibits dynamic behavior, it still possesses favorable data parallel characteristics. For example, we can schedule multiple threads to simultaneously compute read-write intersections between different start instructions, end instructions, and the same intermediate instruction.

This paper proposes DRPC, a parallel data dependence computation method based on LLVM IR in a GPU environment that achieves efficient data dependence computation through CPU-GPU collaboration. Specifically, the contributions include: (a) a CPU-GPU data mapping method for data dependence computation that effectively hides data mapping latency while ensuring scalability; (b) a multi-stream collaborative direct data dependence computation method and a GPU memory-sensitive dual-end collaborative transitive data de-

pendence computation method that addresses coupling relationships between different iterations while ensuring computational accuracy and achieving CPU-GPU collaboration; and (c) a data distribution strategy that considers GPU memory characteristics and data access patterns in dependence computation tasks to further improve memory access efficiency.

2 DRPC Framework

Focusing on three scalability issues in parallel data dependence computation—bandwidth waste from irregular memory access, efficient CPU-GPU collaboration, and thread asynchronization disrupting the dynamic nature of transitive dependence computation—we propose the Data Dependence Relation Parallel Computation (DRPC) framework based on LLVM IR. As shown in [Figure 3: see original paper], DRPC consists of IR characteristic instruction matching and information extraction (Pre_{{{Match}}*{{Extract}}}), data adaptation storage mapping (*Info*{Storage}), direct dependence computation (Direct_{DepCal}), transitive dependence computation (Trans_{DepCal}), parallel optimization mechanisms (Parallel_{Opt}), and data dependence application output (DepOutput_{forApp}).

Pre_{{{Match}}*{{Extract}}} is responsible for feature-guided key instruction matching and extraction of relevant program analysis information. *Info*{Storage} handles data information mapping between CPU and GPU according to the needs of direct and transitive dependence computation. These two components form the foundation of DRPC. After adaptive data mapping, Direct_{DepCal} and Trans_{DepCal} collaboratively complete various data dependence computations. Direct_{DepCal} performs parallel computation of direct data dependencies, while Trans_{DepCal} completes transitive data dependence computation based on the results of direct dependence computation. During parallel computation, Parallel_{Opt} implements further performance optimizations. Finally, DepOutput_{forApp} organizes and outputs data dependence relationships according to application requirements.

Data dependence information is stored in matrix form, with Matrix defined as:

$$Matrix[i, j] \in \{ND, FD, AD, OD, TD\}$$

where i and j represent instruction indices for Inst and Inst , and the values ND, FD, AD, OD, and TD indicate no dependence, flow dependence, anti-dependence, output dependence, and transitive dependence between Inst and Inst , respectively.

3.2 Feature-Guided IR Instruction Preprocessing

Pre_{{{Match}}_{{Extract}}} in DRPC handles IR instruction analysis and preprocessing. First, it filters candidate instructions from IR text based on

feature keywords (alloca, store, load). Then, it extracts information required for data dependence computation according to instruction organization patterns. This work focuses on instruction-level data dependence in IR; for loop structures, we directly extract instruction information and analyze pairwise instruction dependencies.

summarizes the organization patterns of the three instruction types and their descriptions, where OP1 and OP2 represent instruction operands and OPType represents variable data types.

Pre_{{{Match}}*{{Extract}}*' *s main workflow is described in Algorithm 1, where Match(x, y) matches raw IR instruction x against key instructions and returns the instruction type to y. Algorithm 1 processes IR text line by line, calling Match(Inst, Inst*{Type}) to match key instructions (lines 1-2). If matched as alloca, it stores extracted instruction information in var_{arr} (lines 3-5). If matched as store or load, it stores the information in inst_{{{read}}*{{arr}}} or inst*{{{write}}_{{arr}}} (lines 6-9).

3.3 Multi-Stream Collaborative Direct Dependence Computation

DRPC first computes direct data dependencies (flow, anti-, and output dependencies) through Direct_{DepCal} using a multi-stream collaborative approach. [Figure 5: see original paper] illustrates the working mechanism of multi-stream collaborative direct dependence computation.

We use CUDA as DRPC' s GPU parallel computing environment. CUDA' s thread model has three hierarchies: grid, block, and thread. Each grid contains multiple blocks, each block contains multiple threads, and each thread, block, and grid has a unique ID. Threads within the same block can access a shared memory region, while all threads can access global memory. During dependence computation, DRPC partitions data by thread count and uses thread IDs for data indexing, enabling multiple threads to compute data dependencies simultaneously.

3.1 Storage Design

Three vectors—var_{arr}, inst_{{{read}}*{{arr}}}, and inst*{{{write}}*{{arr}}} *—store Write and Read sets for alloca, store, and load instructions. var*{arr} stores variable information from IR, with vector indices corresponding to variable IDs. inst_{{{read}}*{{arr}}} and inst*{{{write}}*{{arr}}} *store memory read/write operation variable information, with indices corresponding to instruction IDs. Read and write variables in the same instruction share the same index in inst*{{{read}}*{{arr}}} and inst*{{{write}}*{{arr}}}; *memory constant reads are assigned the special value -1. [Figure 4: see original paper] shows the instruction information storage for the example in [Figure 1: see original paper]. For the first store instruction, variable %a*

has index 0 in var{arr}, which becomes its variable ID. The store instruction' s index 0 in inst_{{{write}}*{{arr}}} stores %a' s ID (0), while index 0 in inst*{{{read}}_{{arr}}} stores the special value -1 representing constant 1.

3.3 Multi-Stream Collaborative Direct Dependence Computation (Continued)

Two streams, Stream0 and Stream1, execute data copying, kernel execution, and result return in pipeline fashion. For example, after Stream0 completes data mapping, it executes GPU_{{{Direct}}*{{Kernel}}} for direct dependence computation while Stream1 calls Info*{Storage} for data mapping. When Stream0 finishes kernel execution, it returns direct dependence information to the CPU while Stream1 launches GPU_{{{Direct}}*{{Kernel}}}. GPU*{{{Direct}}*{{Kernel}}} computes direct dependencies by intersecting inst*{{{read}}*{{arr}}} and inst*{{{write}}_{{arr}}} as shown in Equation (2), where Inst and Inst represent start and end instructions with indices s and e.

Algorithm 2 describes CPU_{{{Direct}}*{{DepCal}}}, which handles CPU-side operations. *Info*{Storage}(x) performs data mapping via stream x; GPU_{{{Direct}}*{{Kernel}}}(x) executes the GPU kernel via stream x; *Data*{Return}(x, y, z) returns y from stream x to z. For each function f, CPU_{{{Direct}}*{{DepCal}}} first calls *Info*{Storage}(Stream0) to map data, allocating GPU storage d_{{{Read}}*{{Arr}}}, *d*{{{Write}}*{{Arr}}}, and *d*{Matrix} for direct dependence sets (lines 1-2). It then calls GPU_{{{Direct}}*{{Kernel}}}(Stream0) for computation (line 3) and *Data*{Return}(Stream0, d_{matrix}, Depdirect) to return results while starting Info_{Storage}(Stream1) (line 4). Finally, it calls Data_{Return}(Stream1, d_{matrix}, Depdirect) for the next computation cycle (lines 5-6).

Algorithm 3 describes GPU_{{{Direct}}*{{Kernel}}}. Share_W and Share_R are shared arrays that reduce memory latency by caching *d*{{{Read}}*{{Arr}}} and *d*{{{Write}}*{{Arr}}} data. Init(Array, x, y) initializes Array using indices x and y. The algorithm first computes start and end instruction indices s and e from thread and block IDs (line 1), then initializes Share_W and Share_R (lines 2-3), computes direct dependencies using Equation (2), and stores results in *d*{Matrix} (lines 4-12).

3.4 GPU Memory-Sensitive Dual-End Collaborative Transitive Dependence Computation

Based on Direct_{DepCal}' s results, DRPC computes transitive dependencies via Trans_{DepCal}. This stage first locates flow dependence chains. For an instruction sequence Inst , Inst $_1$, ⋯, Inst (j > i), Equation (3) determines whether a flow dependence chain exists between Inst and Inst based on indices i and j.

Flow dependence chain construction is a dynamic process whose length

grows as intermediate instructions are added. Algorithms 4 and 5 describe CPU_{{{Trans}}*{{DepCal}}}* *and* *GPU*{{{Trans}}*{{Kernel}}}*. *Map*{midNumber}(x) maps intermediate instruction Inst ' s index x; GPU_{{{Trans}}_{{Kernel}}}(k) passes parameter k to the kernel.

Algorithm 4 first allocates GPU storage d_{{{Trans}}*{{Mat}}}* *for Depdirect (line 2), processes each intermediate instruction Inst in Smidinst to obtain its GPU-mapped index k, calls GPU*{{{Trans}}*{{Kernel}}}(k) to construct and analyze dependence chains containing intermediate instructions (lines 3-6), and finally returns d*{{{Trans}}_{{Mat}}} to Deptrans (line 7).

Algorithm 5 obtains thread ID threadId and block ID blockId (line 1), initializes shared array Share_k (line 2), and computes transitive dependencies by locating flow dependence chains from blockId (start) to threadId (end) (lines 3-5).

3.5 Parallel Optimization Mechanisms

During Trans_{DepCal}, Parallel_{Opt} addresses heterogenous computing synchronization and data distribution. First, transitive dependence computation requires adding intermediate instructions between start and end instructions to construct dependence chains. Analyzing each new chain depends on results from other chains, so new chain construction must occur after other chains complete computation. To ensure accuracy, dual-end collaboration transfers new chain construction data from CPU to GPU only after current GPU analysis finishes, synchronizing threads and avoiding missed results. Second, considering memory access efficiency, dependence relationships are stored in matrix format in global memory. During data partitioning, DRPC leverages GPU global memory coalescing by strip-mining the matrix and establishing index mapping between intermediate instruction index k and start instruction index. Under this scheme, a warp accesses a contiguous global memory address range as shown in Equation (4), where N represents matrix width and j represents thread ID (also the end instruction index). Contiguous address ranges fully exploit global memory coalescing, reducing access frequency. Additionally, since data involving intermediate instructions is frequently accessed, it is stored in shared memory to improve locality and access efficiency.

3.6 Data Dependence Application Output

After computation completes, DRPC can organize and output data dependence relationships according to application requirements. [Figure 6: see original paper] shows a sample data dependence graph output for Figure 1: see original paper, where elliptical nodes represent instructions organized as simple equations (left side: write variables, right side: read variables), Entry is the entry instruction node, and dashed arcs with arrows represent flow dependencies.

4.1 Experimental Environment

Experiments run on an Intel Core i7 CPU with NVIDIA GeForce GTX 1050Ti GPU, using SPEC 2000 benchmarks with LLVM IR files as input. We selected 254.gap, 183.equake, 186.crafty, and 176.gcc from SPEC 2000. shows statistics for store and load instructions, which account for over 81% of data-dependence-related instructions. By leveraging GPU global memory coalescing and intra-block data sharing with one-to-one thread-to-instruction mapping, DRPC enables a warp to simultaneously access 32 contiguous instructions, improving global memory efficiency. With small instruction counts, fewer threads lead to lower parallelism and less effective latency hiding. With large instruction counts, more active threads better hide latency, increasing both task and resource parallelism.

4.2.1 Time Overhead

presents DRPC' s computation time data, where SDIR and STRA represent serial direct and transitive dependence computation times, and PDIR and PTRA represent parallel algorithm times (including CPU-GPU data transfer). [Figure 7: see original paper] and [Figure 8: see original paper] compare serial vs. parallel times for direct and transitive dependence computation, respectively. Due to order-of-magnitude differences, results are split into (a) and (b) sections. [Figure 9: see original paper] shows speedup data, where DIRS and TRAS represent speedups for direct and transitive dependence computation.

Analysis of [Figure 7: see original paper] shows PDIR outperforms SDIR for all benchmarks, with greater advantages for larger IR files. Direct dependence speedup ranges from 2.29× to 3.48×, as increased active threads and multi-stream collaboration effectively hide data copying latency. In [Figure 8: see original paper], small datasets show minimal STRA-PTRA differences (1.8× speedup) due to insufficient GPU threads and poor latency hiding. For larger datasets in Figure 8: see original paper, coalesced access and shared memory improve efficiency, while more active threads better hide latency, boosting transitive dependence speedup to 4.91×.

4.2.2 Space Overhead

compares space overhead between serial (SDDC) and parallel (PDDC) data dependence computation. SDDC uses only CPU memory, while PDDC' s CPU space includes pinned memory for multi-stream data copying, adding ~0.8% overhead. PDDC also includes GPU memory for parallel computation, with GPU copying overhead reaching at most 4.9% of PDDC total. Since multi-stream allocation only requires space for one function' s instructions, overall SDDC space overhead is close to PDDC.

5 Related Work

Data Dependence Analysis Methods. Johnson et al. proposed a collaborative dependence analysis framework for LLVM that uses specialized algorithms for different dependence scenarios, achieving both accuracy and composability. Gao et al. presented a practical data dependence analysis method that employs different techniques based on array subscript types to address loop normalization limitations. Jiang et al. proposed a dependence analysis method based on exception propagation analysis that constructs system dependence graphs for exception-handling C++ programs, improving analysis accuracy. Nikolaos et al. developed PARTEE for dynamic data dependence between recursive parallel tasks, which analyzes memory traces to determine dependencies. Experiments show PARTEE solves more fine-grained problems than Nanos++ and outperforms Cilk on irregular task dependencies by 103%. Litvak et al. introduced a field-sensitive transitive dependence analysis algorithm that reduces memory consumption by 31% for large struct variables.

Parallel Data Dependence Analysis. Minjang Kim et al. proposed SD3, a scalable dynamic data dependence profiling framework using hybrid pipeline and data parallelism, achieving 4.1× and 9.7× speedups on 8-core and 32-core processors. Yu et al. presented a method that partitions analysis tasks into independent slices, generating subgraphs through parallel analysis that are automatically combined by the compiler. Experiments show this approach reduces analysis time by orders of magnitude for well-known benchmarks. Li et al. developed a lock-free parallel data dependence analysis method that assigns a worker thread to each memory address, achieving 2.1×–2.4× speedup over serial methods.

6 Conclusion

To address scalability issues in IR-based data dependence computation, this paper proposes DRPC, a parallel data dependence computation framework based on LLVM IR. DRPC exploits parallelism in traditional iterative methods, solves data adaptation and synchronization challenges for GPU-based IR dependence computation, and improves task parallelism through intelligent data allocation based on access frequency. Experiments demonstrate that DRPC effectively enhances the scalability of IR-based data dependence computation.

References

[1] Lattner C. The architecture of open source applications: LLVM [EB/OL]. (2012) [2018-8-1]. http://www.aosabook.org/en/llvm.html.

[2] Wang Tao, Han Lansheng, Fu Cai, et al. Static detection model and framework for software vulnerability [J]. Computer Science, 2016, 43(5): 80-86.

[3] Bates S, Horwitz S. Incremental program testing using program dependence graphs [C]//Proc of the 20th ACM SIGPLAN-SIGACT Symposium on Princi-

ples of programming languages. New York: ACM Press, 1993: 384-396.

[4] Binkley D. Using semantic differencing to reduce the cost of regression testing [C]//Proc of Conference on Software Maintenance. Piscataway, NJ: IEEE Press, 1992: 41-50.

[5] Wang Kechao, Wang Tiantian, Su Xiaohong, et al. Key scientific issue and state-art of automatic software fault localization [J]. Chinese Journal of Computers, 2015, 38(11): 2262-2278.

[6] Miao Li, Zhang Dafang. Computing backward slice of EFSMs [J]. Journal of Software, 2004, 15(suppl): 169-178.

[7] Wu You. The design and implementation of dynamic data dependence analysis tool for C program based on LLVM [D]. Changchun: Jilin University, 2016.

[8] Nicholas W. CUDA handbook: a comprehensive guide to GPU programming [M]. Beijing: China Machine Press, 2014: 97-98.

[9] Johnson N P, Fix J, Beard S R, et al. A collaborative dependence analysis framework [C]//Proc of IEEE/ACM International Symposium on Code Generation and Optimization. Piscataway, NJ: IEEE Press, 2017: 148-159.

[10] Gao Nianshu, Zhang Zhaoqing, Qiao Ruliang. Practical data dependence analysis [J]. Chinese Journal of Computers, 1995, 18(4): 258-265.

[11] Jiang Shujuan, Xu Baowen, Shi Liang, et al. An approach to analyzing dependence based on exception propagation analysis [J]. Journal of Software, 2007, 18(4): 832-841.

[12] Papakonstantinou N, Zakkak F S, Pratikakis P. Hierarchical parallel dynamic dependence analysis for recursively task-parallel programs [C]//Proc of Parallel and Distributed Processing Symposium. Piscataway, NJ: IEEE Press, 2016: 933-942.

[13] Litvak S, Dor N, Bodik R, et al. Field-sensitive program dependence analysis [C]//Proc of ACM SIGSOFT International Symposium on Foundations of Software Engineering. New York: ACM Press, 2010: 287-296.

[14] Kim M, Kim H, Luk C K. SD3: A Scalable Approach to Dynamic Data-Dependence Profiling [C]//Proc of IEEE/ACM International Symposium on Microarchitecture. Piscataway, NJ: IEEE Press, 2010: 535-546.

[15] Yu Hongtao, Li Zhiyuan. Multi-slicing: a compiler-supported parallel approach to data dependence profiling [C]//Proc of International Symposium on Software Testing and Analysis. New York: ACM Press, 2012: 23-33.

[16] Li Zhen, Jannesari A, Wolf F. An Efficient Data-Dependence Profiler for Sequential and Parallel Programs [C]//Proc of Parallel and Distributed Processing Symposium. Piscataway, NJ: IEEE Press, 2015: 1037-1046.

Note: Figure translations are in progress. See original paper for figures.