Locality Optimization for Program Instances

Claudia Leopold

Fakultät für Mathematik und Informatik
Friedrich-Schiller-Universität Jena
07740 Jena, Germany
claudia@informatik.uni-jena.de

Abstract. The degree of locality of a program reflects the level of temporal and spatial concentration of related data and computations. Locality optimization can speed up programs by reducing the communication costs. A possible human approach to locality optimization is to consider several small program instances of a given program, find optimal or close to optimal mappings of data and computations for the program instances, and generalize them to the program.

This paper suggests the use of this approach in a semi-automatic locality optimization method. Emphasis is given to the phase of optimizing the program instances. We introduce a fuzzy objective function that reflects the degree of locality of a program instance, and show its advantages over a crisp function such as communication costs. We provide a framework for applying several optimization techniques and describe an implementation that uses local search. The paper refers to a two-level memory hierarchy.

1 Introduction

The communication costs of programs largely depend on the temporal and spatial mapping of data and computations to processors and memory blocks. Locality is a property of programs that reflects the level of concentration of accesses to the same datum or to data that are stored together, e.g. in the same cache line.

Locality optimization can speed up programs significantly. This holds true for both shared and distributed memory parallel computers, and also for sequential computers or nodes of parallel computers with respect to their local memory hierarchy. The most basic case is a sequential memory hierarchy, where a temporal mapping of computations and a spatial mapping of data to memory blocks (e.g. cache lines) are to be found.

This paper refers to the basic case and considers a two-level model, where memory is divided into a fast module with limited capacity M and a slow module with potentially infinite capacity. Data is stored in blocks of fixed size B. Whenever some datum is accessed, its block is moved to the fast memory (if it is not already there). If the fast memory is full, the least recently used block is replaced. The communication cost of a program is the number of block transfers between the memory levels. We optimize the locality of programs by reordering the program statements (code transformations) and reassigning the data to memory blocks (data transformations).

The paper is organized as follows. In Section 2, a semi-automatic method for locality optimization is motivated and described. It is based on the automatic optimization of program instances, followed by a manual generalization of the solutions. Section 3 deals with the optimization problem for program instances. It starts with a formalization of the problem, in Section 3.1. Next, in Section 3.2, we introduce two possible objective functions for locality: a crisp function that is based on communication costs and a fuzzy function that is based on the intuitive notion of locality. We argue for the superiority of the fuzzy function. Section 3.3 outlines how we strive at finding regular solutions to support the generalization phase, and Section 3.4 gives some details on the implementation. Finally, Section 4 is devoted to experimental results, concerning both the optimization algorithm for program instances and the semi-automatic method.

2 The Semi-Automatic Method

Locality optimization resembles well-investigated combinatorial optimization problems. The code and data transformation aspects can be compared with the travelling salesman problem and with graph partitioning, respectively. There exist many powerful optimization techniques for these problems, including biologically inspired techniques such as genetic algorithms and neural networks. Unfortunately, the direct application of these techniques to locality optimization is often impeded by the fact that the sets of operations and data are not given explicitly. Typically, the same program may be run with different input sizes implying different sets of operations and data. Even if the input size is known, the sets of operations and data are often so large that the optimization techniques would be too prohibitively slow.

A possible human approach to locality optimization is to consider several small program instances of a given program, find optimal or close to optimal mappings of data and computations for the program instances, and generalize them to the program. We suggest using this approach in a locality optimization method that is currently semi-automatic.

The user has to specify one or several program instances, by fixing parameters of the input program. A program instance appears as a set of statement instances (SI), where a SI is an elementary or complex instantiated statement. Each SI must be carried out exactly once. An example of a program instance is given in Figure 3a. The user also has to decide on the granularity of the SIs. Sequences of operations with heavy internal control dependencies should be taken as complex SIs. For efficiency reasons, the input size of the program instances must be fixed at a small value. The user also has to specify memory hierarchy parameters M and B that are appropriate to the size of the program instance, i.e., that reflect the proportions of the real case correctly. In fact, a relatively wide range of parameters will work well. Additionally, the input must comprise a data dependence graph. Since we have unrolled loops, it is a directed acyclic graph, referred to as dag. It can be easily and automatically constructed.

Now, we have an optimization problem that is accessible for the various optimization techniques. Since the program instances are small, the complexity is manageable. A computation and data mapping for the program instance is derived fully-automatically. In this paper, we refer to the technique of local search, but other techniques could be adapted as well.

Finally, the mapping found for the program instance has to be generalized into a mapping for the program; in particular, it is the structure of the mapping that is generalized. The generalization is done manually, with some automatic support that consists in preferring simple, regular mappings in the output of the optimization.

The semi-automatic method can handle those programs where structurally identical mappings perform well for all program instances. Independent of its use in the semi-automatic method, an optimization technique for program instances can predict the quality of given mappings: A mapping performs well for a program, if, applied to program instances, it is as comparatively good as a mapping found by an optimization technique for program instances.

3 The Optimization Problem for Program Instances

3.1 Formal Statement of the Problem

Input:

- Sets of statements and data $S = \{s_1 \dots s_{|S|}\}$ and $D = \{d_1 \dots d_{|D|}\}$
  For readability, we use the notion statement as an equivalent for SI.
- dag (directed acyclic graph with nodes marked by $s_1 \dots s_{|S|}$)
- $M, B \in \mathbb{N}$ with $M, B \geq 2,\ B \mid M,\ B \mid |D|$
- For each $s \in S$, a sequence of data $d_1(s) \dots d_{k_s}(s)$ $(d_i(s) \in D)$
  The sequence comprises the data that is accessed by s in this order. The information is used for locality optimization purposes.
- For each $s \in S$, a characterization of the operation carried out and a sequence $z_1(s) \dots z_r(s)$ of indices appearing in the name of s. The information is used for regularity optimization purposes.

Output:

- Statement ordering, i.e., permutation *Sched* with $\mathit{Sched}(i) = s$ iff statement s is placed at the i-th position in the statement sequence $(s \in S,\ i = 1 \dots |S|)$
- Data assignment, i.e., function DB with $DB(d) = b$ iff datum d is assigned to memory block b $(d = d_1 \dots d_{|D|},\ b \in \{1 \dots M/B\})$

Constraints:

- $\mathit{Sched}(i_1) = s_1$ and $\mathit{Sched}(i_2) = s_2$ $(i_1 < i_2)$ is permitted only if there is no path from s_2 to s_1 in the dag
- $|\{d \in D \mid DB(d) = b\}| = B,$ for all $b \in \{1 \dots M/B\}$

An objective function will be added in the following sections.

3.2 Objective Functions for Locality

Since locality optimization aims at reducing communication costs, an obvious objective function is the number of block transfers required for the simulated execution of the program instance in the two-level memory model. For simplicity, only loads are counted, and we do not distinguish between reads and writes. The corresponding function is denoted by OFL_{cc}.

We favour another objective function, OFL_{loc}, that is based on the rather fuzzy definition for the degree of locality given in the introduction. Applied to the two-level model, the definition of the degree of locality is connected to the level of concentration of accesses to the same memory block during program execution. A higher level of concentration of the accesses implies a higher degree of locality. In the following, we introduce a quantitative measure for the degree of locality based on this definition.

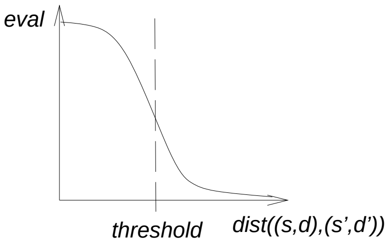
Let *reuse-tuple* denote a pair of accesses such that the memory block brought into the fast memory by the first access may be reused by the second access if it still resides in the fast memory. Formally, it is a tuple $((s,d),(s',d'))$ with $s,s' \in S,\ d,d' \in D,\ (s,d) \neq (s',d')$ and $DB(d) = DB(d')$. Furthermore, the access of s to d must precede the access of s' to d', and there must not be another access to $DB(d)$ between. Let $dist((s,d),(s',d'))$ denote the number of accesses between the access of s to d and the access of s' to d'.

From the intuitive meaning of the notion, a quantitative measure for the degree of locality must satisfy the following requirements:

- the degree of locality is higher the more reuse tuples are scheduled closer together.
- the contribution of a particular reuse tuple to the degree of locality takes the existence of a threshold into account with

$$\begin{array}{ll} dist((s,d),(s',d')) \le threshold \rightarrow & \text{block } DB(d') \text{ is still in fast memory} \\ & \text{at access } (s',d') \\ dist((s,d),(s',d')) > threshold \rightarrow & \text{block } DB(d') \text{ must be loaded anew} \\ & \text{for access } (s',d') \end{array}$$

- the function *eval* that determines the contribution of a particular reuse tuple to the degree of locality must have about the following shape



We did extensive experiments in which the locality of two statement sequences was compared intuitively on one hand, and with a candidate objective function on the other. Thereafter, we decided to define OFL_{loc} by

$$\sum_{\substack{\text{reuse tuples}\\((s,d),(s',d'))}} eval(dist((s,d),(s',d')))$$

with

$$eval(dist) = \begin{cases} 1 - 1/(threshold + 2 - dist) & dist \leq threshold \\ 1/\sqrt{dist - threshold + 4} & dist > threshold \end{cases}$$

The same threshold is used for all reuse tuples. The threshold must be estimated, e.g. from an optimized program suggested by compiler optimizations. The optimization procedure can also be repeated with different thresholds. The function OFL_{loc} has several advantages over OFL_{cc}:

- The range of OFL_{loc} is continuous. Hence, it can better differentiate between statement sequences of about the same quality. This is advantageous for optimization methods that are based on stepwise refinement.
- The function OFL_{loc} can be evaluated faster. In particular, it can be quickly estimated by considering only those reuse tuples that are placed closely.
- The function OFL_{loc} can tolerate inaccuracies in the specification of the optimization problem. The objective function value obtained with the fuzzy "the-closer-the-better"-principle of OFL_{loc} depends much less on the concrete value of M than the value obtained with the crisp "either-close-enough-or-not"-principle of OFL_{cc}. The same holds true for other inaccuracies in the problem specification, including arbitration in the specification of the order of accesses within a statement, as well as limited associativity of caches. These inaccuracies may falsify the communication costs predicted by OFL_{cc}. Since inaccuracies cannot be avoided in the semi-automatic method, the ability to tolerate them is an important argument for OFL_{loc}.

3.3 Combination with an Objective Function for Regularity

To support the generalization phase of the semi-automatic method, we strive at finding regularly structured solutions. In the current implementation, locality has absolute priority over regularity. The program instance is first optimized for locality, exclusively. Afterwards, the output statement sequence is improved w.r.t. regularity. A modification is said to improve the sequence if it improves the regularity and does not decrease the degree of locality (referring to either OFL_{cc} or OFL_{loc}). I.e., in the last phase of the optimization, we combine the objective function for locality with an objective function for regularity. Details on this function are omitted. Briefly, it identifies loops in the statement sequence, and determines a degree of regularity as the sum of valuations of the identified loops.

3.4 Local Search and Implementation Details

Our implementation uses local search to solve the above stated problem. Local search is a well-known general heuristic solution approach to combinatorial optimization problems [3]. The algorithm starts with a random statement ordering and data assignment not violating constraints. Then, it repeatedly improves the current solution w.r.t. the objective function, via simple transformations such as moving statement groups or swapping the block assignment of two data. The algorithm stops when the transformations under consideration cannot further improve the solution anymore.

In the implementation, we decided to modify OFL_{loc}. Recall that in the definition of reuse tuples, the statements are required to access the same *block*. Alternatively, we could require them to access the same *datum*. In the implementation, the objective function value is determined as the weighted sum of the objective function values obtained with these differing definitions. The weight for the datum-based function value is high at the beginning, and it is reduced as the search goes on.

4 Experimental Results

We applied the optimization procedure to several instances of matrix multiplication, matrix transpose and mergesort programs, as well as of nested loop programs taken from [6] and [9]. We obtained the following results.

Result 1: *With respect to finding computation mappings for small program instances (assuming the data assignment to be fixed), the optimization method improves on the optimization method of [9] (which is a typical compiler optimization method).*
Some measurements are given in Table 1. In Table 1, N is the input size, i.e., we consider e.g. $N \times N$ matrix multiplication. Data distribution $k \times l$ denotes that a memory block is formed by $k \times l$ subblocks of the matrices under consideration. The columns "OFL_{loc}", "OFL_{cc}" and "reference" give the communication costs of the optimized program instance, when the optimization was done with OFL_{loc}, with OFL_{cc}, or with the reference optimization method from [9], respectively. The values refer to the minimum achieved over five random runs of our algorithm, with a reasonable value for the threshold. The column "time" gives the processor time in seconds, referring to a C implementation on a 200MHz Pentium PC, for OFL_{loc}.

Result 2: *With respect to finding both data and computation mappings for small program instances, the optimization method improves on the compiler optimization method of [6].*
Some typical measurements are given in Table 2. Column "OFL^1_{loc}" refers to OFL_{loc} as described in Section 3.2, "OFL^2_{loc}" refers to the modified objective function, as explained in Section 3.4. The time measurements refer to the modified objective function.

program	N	M	B	data distr.	OFL_{loc}	OFL_{cc}	reference	time
[9] Fig. 3	8	4	2	1 × 2	7	7	15	0.4
[9] Fig. 3	12	6	2	1 ×2	11	16	33	418
transpose	8	32	4	2 ×2	32	32	32	9.2
transpose	8	12	2	1 ×2	64	67	64	7.0
multiplication	4	12	2	1 ×2	36	36	40	16
multiplication	4	12	4	2 ×2	17	17	20	13.0
mergesort	64	20	4	1×4	120	122	-	11.2

Table 1. Measurements concerning Result 1.

prog.	N	M	B	OFL^1_{loc}	OFL^2_{loc}	OFL_{cc}	reference	time
[9] Fig. 3	8	4	2	7	6	7	15	5.2
[6] Fig. 2A	2	6	2	7	7	7	10	0.02
[6] Fig. 2F	$N_1, N_2, N_3 = 6, 6, 2$	10	2	39	38	40	38	9.4
transpose	4	16	2	16	16	16	16	0.2
multiplication	4	12	2	52	37	62	40	114
mergesort	32	6	2	144	144	144	-	17.0

Table 2. Measurements concerning Result 2.

Result 3: *The semi-automatic method can speed up programs significantly, even in comparison with compiler optimizations. It can also speed up programs where current compiler optimizations are not applicable. The inclusion of regularity aspects supports the generalization phase.*
The semi-automatic method is illustrated with the example of mergesort, in Figure 3. We refer to a simple non-recursive program for mergesort that alternatingly uses two arrays and assume a fixed block-wise data distribution. The programmer has to fix the input size and the granularity of the operations, as well as M and B. Observing that, in real cases, the fast memory can hold only part of the data, he may e.g. decide for $N = 8$, $M = 8$ and $B = 2$. Because of heavy internal control dependencies, he will decide to equate statements to the complex operations of merging two runs. The corresponding program instance is given in Figure 3a. Now, the optimization method for program instances is applied. It starts with a random feasible solution, e.g. with that in Figure 3b. Optimizing for locality only, the optimization method may e.g. come up with the statement sequence given in Figure 3c. Appending a regularity optimization phase, the statement sequence may be improved into the statement sequence given in Figure 3d. In Figure 3d, it is easier to identify the structure of the solution. The programmer will probably repeat the optimization for more program instances. Then, it is his task to generalize the solutions found for the program instances into a solution for the program. In the case of mergesort, he would come up with a recursive program.

a) Merge [0..0] with [1..1]
Merge [2..2] with [3..3]
Merge [4..4] with [5..5]
Merge [6..6] with [7..7]
Merge [0..1] with [2..3]
Merge [4..5] with [6..7]
Merge [0..3] with [4..7]

b) Merge [4..4] with [5..5]
Merge [2..2] with [3..3]
Merge [0..0] with [1..1]
Merge [0..1] with [2..3]
Merge [6..6] with [7..7]
Merge [4..5] with [6..7]
Merge [0..3] with [4..7]

c) Merge [6..6] with [7..7]
Merge [4..4] with [5..5]
Merge [4..5] with [6..7]
Merge [0..0] with [1..1]
Merge [2..2] with [3..3]
Merge [0..1] with [2..3]
Merge [0..3] with [4..7]

d) Merge [0..0] with [1..1]
Merge [2..2] with [3..3]
Merge [0..1] with [2..3]
Merge [4..4] with [5..5]
Merge [6..6] with [7..7]
Merge [4..5] with [6..7]
Merge [0..3] with [4..7]

Fig. 3. The semi-automatic method on the example of mergesort.

We have run the input and optimized programs for mergesort with several realistic input sizes (up to 10^6) on a DEC 3000 model 600 and observed that the optimized program improves on the input program by about 40% w.r.t. primary data cache misses (measured with pfm), and by about 18% w.r.t. user time. For a simple nested loop program ([9] Fig. 3), the semi-automatic method improved on the compiler optimized program given in [9] by about 18% w.r.t. cache misses and by about 15% w.r.t. user time (for input sizes up to 10^4 and several tile sizes). In both cases, the programs were coded in C and compiled with gcc. Mergesort is an example of a program where current compiler optimizations are not applicable.

Result 4: OFL$_{\mathbf{loc}}$ is superior to OFL$_{\mathbf{cc}}$.

The superiority with respect to solution quality can be seen in Tables 1 and 2. We also observed that OFL_{loc} is more tolerant towards inaccuracies in the input.

5 Related Work

A lot of work has been done on compiler optimizations for locality, referring to both two-level sequential memory hierarchies [6] [9] and shared or distributed memory parallel architectures [1] [2]. In the past, data and code transformations were typically considered in separation but since recently there is a tendency to combine them. All compiler optimizations we are aware of have in common,

- they aim at regular computations characterized by nested for-loops with simple loop bounds that access arrays via affine index expressions, as they are typically found in scientific computing.

- they consider a restricted set of transformations.
- they use heuristics that are tied to the particular set of transformations.
- the transformations are applied to the *program*, assuming that the same transformations will perform well for all program *instances.*

In contrast to current compiler optimizations, the semi-automatic method suggested in this paper may find non-standard transformations, since it does not restrict the set of transformations but searches the complete space of permitted mappings. Furthermore, its applicability is not restricted to nested loop programs with affine index expressions.

Other related work is the consideration of locality issues in algorithm/program design [7], [8]. Experience shows that locality considerations in algorithm/program design are time-consuming and complicated. The semi-automatic method may save human work, since the programmer does not have to *find* the mapping anymore, he only needs to *recognize* it. Still, locality optimization by the programmer is more general.

Locality issues are also considered in connection with task scheduling [4], [5]. Task scheduling typically concentrates on applications where the number of tasks is rather small such that the real set of tasks can be handled by heuristic techniques.

References

1. J.M. Anderson, S.P. Amarasinghe, M.S. Lam: Data and Computation Transformations for Multiprocessors. *Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pp. 166-178, 1995
2. M. Cierniak, W. Li: Unifying Data and Control Transformations for Distributed Shared Memory Machines. *Proc. ACM SIGPLAN'95 Conf. on Programming Language Design and Implementation*, pp.205-217, 1995
3. Y. Crama, A.W.J. Kolen, E.J. Pesch: Local Search in Combinatorial Optimization. *Artificial Neural Networks*, LNCS 931, pp. 157-174, 1991
4. K. Dussa-Zieger: Configuration, Mapping and Sequencing by Genetic Algorithms. *Proc. Int. Workshop on Approximate Reasoning in Scheduling*, ICSC Press, pp. 11-17, 1997
5. H. El-Rewini, T.G. Lewis, H.H. Ali: Task Scheduling in Parallel and Distributed Systems. Prentice Hall, 1994
6. M. Kandemir, J. Ramanujam, A. Choudhary: A Compiler Algorithm for Optimizing Locality in Loop Nests. *Proc. ACM Int. Conf. on Supercomputing*, pp. 269-276, 1997
7. A. LaMarca, R.E. Ladner: The Influence of Caches on the Performance of Sorting. *Proc. ACM SIGPLAN Symp. on Discrete Algorithms*, pp. 370-379, 1997
8. J.S. Vitter, E.A.M. Shriver: Algorithms for Parallel Memory I: Two-Level Memories. *Algorithmica*, Vol. 12, No. 2/3, pp. 110-147, 1994
9. M.E. Wolf, M.S. Lam: A Data Locality Optimizing Algorithm. *Proc. ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation*, pp. 30-44, 1991