Automatic Parallelization and Minimization of Communication Costs of Program Instances

Michael Süß
michaelsuess@gmx.de
University of Leipzig
Faculty of Mathematics and Computer Science

Supervisor:	Dr. Claudia Leopold
Category of Work:	Diploma Thesis
GI department:	Technische Informatik und Architektur von Rechensystemen (TI)

Abstract

This paper describes an approach called *semi–automatic method for locality optimization* to support parallelization. Our main contribution is a local search algorithm that automatically parallelizes program instances by distributing statements onto a number of processors, and optimizes for cache locality on individual processors. The algorithm is based on an objective function that simulates memory–access times. Experimental results indicate that the modified semi–automatic method is able to optimize program instances for parallel execution, while simultaneously fulfilling statement dependencies.

1.Introduction

During the last decade, high-performance computing has seen a tremendous increase in speed and efficiency. While faster processors, memory subsystems, storage media and buses have done their part to make this success story possible, the heavy use of *parallelization* was the key to achieving these speedups. And yet there is no end in sight, because the demand of computing power is rising steadily, since problems like weather prediction, geological simulations or calculations of astronomical bodies in space (to name just a few) are using more accurate and therefore numerically expensive models. There are disadvantages and problems to be considered with parallelization, though. Sequential programs need to be rewritten, or new ones must be designed, to make use of the available parallel computing power. Usually, this task is accomplished by specialists with years of experience in the field of parallelization. While the introduction of standards like *OpenMP* or *MPI* has made that work easier, it remains a difficult and time-consuming task to write efficient, clean and correct parallel code.

A possible way-out is *automatic parallelization*. Although various compilers for parallel computers have become mature, there are still limitations. Especially in the field of scientific computing much can be gained by hand-optimizing an algorithm for parallelism.

Pretty much the same thing can be said about the treatment of locality. Designing algorithms and compilers with locality in mind is an important, and yet difficult task. Many compilers in use today at least claim to optimize for some kind of locality. But still, writing a program in which the compiler is able to recognize all possible instances of locality remains a challenging task.

With these considerations in mind, the *iblOpt* programming tool has been designed and implemented. It provides a relatively easy way to discover variants of algorithms with a high degree of locality. The thesis presented here aims at enhancing the tool to support a unified approach to automatic parallelization and locality optimization [Sue03].

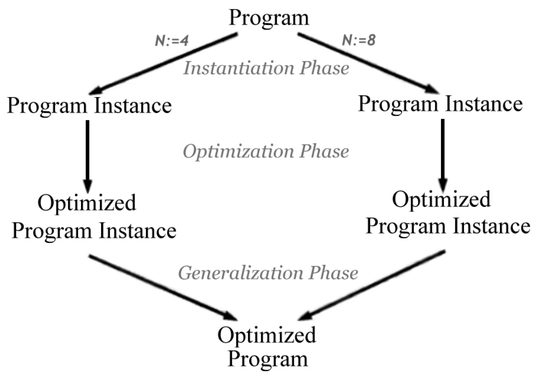
Like *iblOpt*, the extended tool is embedded in a *semi-automatic method,* explained in more detail in section 2. Section 3 describes our extensions to the tool, while section 4 provides experimental results. A summary and outlook are closing this article in section 5.

2.The Semi-Automatic Method and the *iblOpt* Programming Tool

This section describes the *semi-automatic method* and its main component, the *iblOpt* programming tool. The goal of the semi-automatic method is to provide an intermediate alternative between expensively hand-optimizing parallel code, and relying on a fixed set of transformations implemented in parallelizing compilers. It was first introduced as a method for locality-optimization by Leopold in [Leo98].

A coarse description of the method, adapted for automatic parallelization, follows:

1. Identify performance-critical parts of the target application
2. Transform them into program instances, by fixing parameters at small values (*instantiation phase*)
3. Let *iblOpt* find an optimum distribution of statements across processors on an *SMP*-system for the program instance (*optimization phase*)
4. Generalize the solution to the original program (*generalization phase*)



A graphical representation of this method is given in the figure above. In there, all phases are outlined with lighter colors, and *N* is used to indicate the size of the input parameters. More details on the semi-automatic method can be found in [Leo98].

3.Conceptual Work

3.1.Definitions

The main contribution of the thesis summarized in this paper is a local search algorithm that distributes a *program instance* across a predefined number of processors. In our context, the term *program instance* (*PI*) is defined as a set of *statement instances*, together with functions such as:

- a function *SP*: S → N, which assigns each *statement instance* to a processor
- a function *Sched*, which assigns each *statement instance* to a position in the schedule of its currently assigned processor

A *statement instance* (*SI*) is an indivisible unit, which is characterized by the data items it accesses.

3.2.The Local Search Algorithm

The generic approach of local search has been adapted to the needs of our specific problem domain as follows:

- optimum solutions are defined as program instances that perform as fast as possible on a target *SMP*-architecture
- all candidate solutions must obey the dependencies between its statement instances
- an initial random solution is obtained roughly the same way as in the original tool [Leo98]
- to any given solution, a feasible neighbor is constructed in the following way:
    1. On an arbitrary processor *p*, select a number *k* of successively scheduled SIs.
    2. Move the selected SIs to an earlier position in the schedule of the same processor *p*, where no dependencies for any SIs are violated.
    3. Move the selected SIs to an arbitrary position on a different processor *q*, where no dependencies for any SIs are violated.

4. If possible and allowed by the statement dependencies, do the same steps as in one of the two previous points and additionally move k successively scheduled SIs, taken from right before the insertion point, to the position where the former SIs came from This effectively makes the whole transaction an exchange operation (consisting of two moves).

Either point 2 or 3 are executed, never both of them in one iteration.

- all feasible solutions are evaluated for each iteration of the local search algorithm and the best one is chosen as the basis for the next iteration-step, until no better solution can be found
- the objective function used by the local search algorithm is described in section 3.3

3.3. The Objective Function

For our specific problem domain, the objective function *o(f)* maps a PI *f* into the real numbers, thereby indicating solution quality. We are interested in solutions that evenly distribute all input SIs across the given number of processors. But how is a good distribution distinguished from a bad one? In the world of parallel processing, probably the most widely used measure for performance is *wall-clock time*. Above all, we are interested in solutions that perform as fast as possible on the available number of processors. Therefore, our objective function has a simulation component that calculates, among other things, an approximation of the time any given solution (PI) needs to access all required data through the memory subsystem (short: *memory-access time*). The solution with the lowest memory-access time across all processors wins.

However, there are times when many solutions with an equal memory–access time have to be evaluated. Therefore, a secondary objective function was introduced. It calculates the sum of the memory-access times, across all processors (short: *accumulated memory-access time*). While the primary objective function is only able to detect improvements that directly affect the slowest unit, the secondary objective function notices improvements of the memory–access times across all processors. Since we want all units to perform as fast as possible (and the slowest processor must not remain that way after the next iteration of the local search), the *accumulated memory–access time* is also a well suited candidate for an objective function.

A tertiary objective function was added during the course of our experiments, because sometimes the two functions described above were not sufficient to single out a best solution for an iteration. Naturally, the statement instances should be distributed across the processors as evenly as possible, with no single unit being significantly busier than another. The well known mathematical reference number *variance* proved to be able to assure this and was therefore used as a tertiary objective function.

A wrapper function was written to compare two possible solutions, using the three objective functions. Only when the primary function returned an equal result for both solutions, the secondary one was evaluated. The tertiary objective function was used as a last resort, when the primary and secondary functions could not decide on a best solution.

The simulation component presently approximates wall-clock times on an SMP-architecture, with the following cache characteristics: *fully-associative, least recently used, write back, write allocate, write invalidate* and *cache coherent*.

4. Results

For our thesis, three sequential sample programs have been parallelized to show the potential of the semi-automatic method and the *iblOpt* programming tool. These were a matrix transposition, a matrix multiplication and an iterative solution of the laplace equation. All examples were evenly distributed across processors; the results for the parallel version of the laplace equation solver are shown in Table 1.

Input size Processors	1000 2	2000 2	4000 2	1000 4	2000 4	4000 4	1000 8	2000 8	4000 8
W.-clock t.	4,85	19.35	79,91	2,44	10,27	42,28	1,32	5,84	23,68
Ref. w.-clock t.	9,88	38,24	178,38	9,88	38,24	178,38	9,88	38,24	178,38
Speedup	2,04	1,98	2,23	4,05	3,72	4,22	7,48	6,55	7,53

Table 1: Timing results for the parallel laplace equation program

The abbreviation *W. clock t.* stands for wall-clock time and *Ref. w. clock t.* stands for reference wall-clock time; that is for the wall-clock time of the sequential version. All times are in seconds and were measured on the *Hessischer Hochleistungsrechner* [Sch03].

5.Conclusions

5.1.Summary

The main accomplishment of the thesis presented here [Sue03] is the enhancement of the semi–automatic method and the *iblOpt* programming tool to include automatic parallelization. Like *iblOpt*, the enhanced tool uses a local search approach with a specifically designed objective function that is based on three components: memory–access time, accumulated memory–access time and variance of the distribution of statements across processors.

To evaluate the objective function, it became necessary to design and to implement a runtime simulator component, which was able to accurately simulate the memory–accesses performed by each statement on every processor, while at the same time keeping track of corresponding cache–contents. Using this functionality, it was possible to reliably calculate all necessary components of the objective function.

The second contribution of the thesis is the actual algorithm used for automatic parallelization in *iblOpt*, which was adapted for our task from the original locality optimizing solution. It contains the search for a specially constructed neighborhood, and is executed repeatedly for better results. It was shown that this algorithm was able to perform automatic parallelization on a number of sequential programs.

5.2.Outlook

Improvements and further work could be done on several possible subsystems of the *iblOpt* programming tool. Performance improvements of the search time for solutions would be the first sector that came to our mind. Presently, the local search algorithm evaluates far too many possible neighbors, before reaching a conclusion about what move or exchange operation to perform next. Performance also suffers from the runtime simulator function, which presently needs to be recalculated every time a neighbor is evaluated. An incremental solution that builds upon the performance measures obtained for the present solution would be much preferred.

Moving away from performance issues, there are a couple of areas where further work could start as well. Regularity optimization (explained in [Leo01]) is still not implemented for our parallelizing part of *iblOpt*, although this could probably be changed relatively fast by adapting the functionality of the single processor case. This change has the potential to greatly help the programmer during the generalization phase, where it is often quite difficult to recognize patterns from the output PIs.

The biggest improvements of the results could probably be achieved by not only optimizing statements for locality, but also the data distribution. This is implemented in the original *iblOpt* programming tool and documented e.g. in [Leo98], but has not been ported to our new parallelizing component yet. A whole new class of solutions could possibly be reached then.

Further research could also be done while changing fundamental assumptions. More cache characteristics (e.g. *write-update* or *write-through*), different memory organizations (e.g. *NUMA*) or even whole new classes of parallel architectures (e.g. *clusters*) would be possible targets for optimization using the semi-automatic method.

Literature:

[Leo01] Claudia Leopold. Structuring statement sequences in instance-based locality optimization. Future Generation Computer Systems, 17(4):425–440, 2001.

[Leo98] Leopold, Claudia: Arranging statements and data of program instances for locality. Future Generation Computer Systems, 14(5 6):293 311, 1998.

[Sch03] Andreas Schönfeld. Hessischer Hochleistungsrechner - HHLR. http://www.tu-darmstadt.de/hrz/hhlr/, 2003.

[Sue03] Michael Süß. Automatic Parallelization and Minimization of Communication Costs of Program Instances. Diploma Thesis, October 2003