An Overview of Implicit Parallelization

Russell Harmon

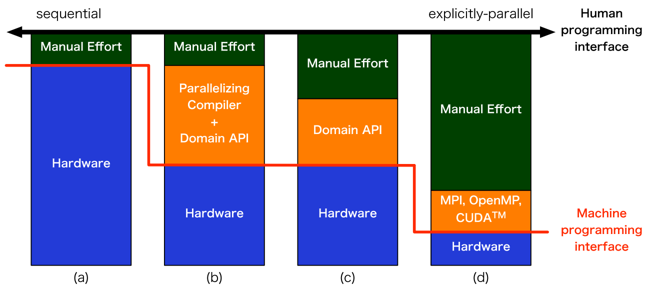


Fig. 1. Programmer Effort[1]

Abstract

In this paper we present the concept of *implicit parallelism*, a method by which a single-threaded program may be converted into a multi-threaded program either through *(i)* a compiler extension *(ii)* a recompiler *(iii)* run-time dynamic recompilation *(iv)* hardware supported parallelism. This enables an application to take advantage of multi-core or multi-processor systems without intricate knowledge of how parallel programming works.

I. Introduction

Parallelization is the act of taking a series of operations and modifying them in such a way as to enable them to be run in *parallel* as opposed to *sequentially*. Although this can provide large performance benefits, with the existing methods of parallelization it can be an error prone and difficult process. A programmer must understand the intricate details of synchronization, synchronization primitives and the non-deterministic nature of a scheduler. There has been a great deal of research into hiding these complexities from the programmer, but due to *(i)* the overhead introduced by parallelization *(ii)* the complexity *(iii)* the tendency for error of parallelization, few efforts have seen success. Of the few that have seen success, one such effort is with genetic algorithms [1].

The design of the common microprocessor has been moving to that of multiple cores since the late nineties [3]. As processors become more and more capable of performing many tasks in parallel, software developers will need to begin to write software which takes advantage of that.

II. Existing Approaches

In this paper, we try to explore the possibility of converting single-threaded applications into multi-threaded applications *automatically*. In the practice of this, five general problems are encountered:

- Different programs are parallelizable to different extents.
- In programs which are parallelizable, work must be done to ensure that the overhead introduced by threading does not overcome the speedup inherent to parallelization [2].
- In languages with lazy evaluation, it is difficult to determine which computational tasks will perform the actual work needed [2].
- Even if the source code for a piece of work appears pure, its compiled implementation may contain side effects (e.g. to perform memoisation) [2].
- IO or updates to mutable storage requires additional handling.

Figure 1 (b) depicts the approach of implicit parallelization with respect to the other possible approaches of parallelization.

Parallelization is not a new concept. There exist a number of approaches which are already successful in the world of software development. The closest current approach to fully automatic parallelization is the approach taken by projects like OpenMP [3] and CUDA [4]. The approaches taken fall into three different classifications shown in Figure 1 (a, c, d). Figure 1 (a) depicts the average multi-threaded application where the user uses synchronization techniques explicitly, Figure 1 (c) depicts domain-specific programming APIs like CUDA, and Figure 1 (d) depicts hinted programming APIs like OpenMP.

The traditional way of parallelizing an application (fig. 1 (a)) is with explicit use of *synchronization primitives*. An application programmer explicitly uses system calls to start and stop threads, and uses primitives such as semaphores to ensure that no two threads can access the same shared memory at the same time.

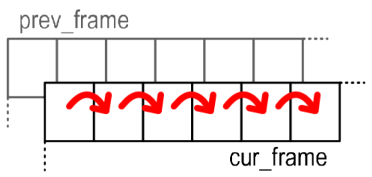
There has been research into a number of possible methods of automatically parallelizing an application. This paper only discusses a subset of the methods currently being investigated.

The approach taken by OpenMP (fig. 1 (d)) and CUDA (fig. 1 (c)) is that of employing a *domain specific API*. This API hides all the complexity of parallelization from the programmer by doing one or both of the following:

- Providing high level function calls for performing complex tasks such as rendering a complex image on screen, the innerworkings of which is parallelized (CUDA).
- Providing annotations by which the API is able to modify the annotated code to be parallelized (OpenMP).

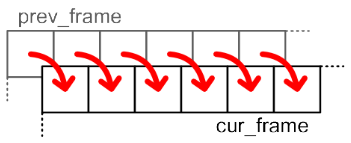
[1]Figure taken from Hwu et al. [3]

```
for (i = 0; i < num_blocks; i++)
  Vector guess = i ?
    (cur_frame[i-1].best_vector) : (0,0);
  cur_frame[i].best_vector =
    GetMatch(cur_frame[i], prev_frame, i, guess);
```



(a) Guess vectors are obtained from the previous macroblock.

```
user_assert(not_overlapping(cur_frame, prev_frame));
for (i = 0; i < num_blocks; i++)
  Vector guess = prev_frame[i].best_vector;
  cur_frame[i].best_vector =
    GetMatch(cur_frame[i], prev_frame, i, guess);
```



(b) Guess vectors are obtained from the corresponding macroblock in the previous frame.

Fig. 2. MPEG-4/H.263 Encoder Motion Estimation Example and Dependence Visualization.[3]

A. CUDA

CUDA is Nvidia's programming API for executing code on their GPUs. This API provides highly parallelized function calls that are designed to be called by a single thread to perform a task. Silberstein, Schuster, Geiger, Patney, and Owens [4] talk about using CUDA to calculate sum-products. This is distinctly different than hinted approaches such as OpenMP in that the actual loop code is not written by the developer.

B. OpenMP

OpenMP is a C based API designed so that you provide hints via the pragma preprocessor macro to tell OpenMP to parallelize the loop that follows. This is distinctly different than functional approaches such as CUDA in that the actual loop code is still written by the developer.

III. Parallelizable Algorithms

There is one change that application developers will need to make. When developing an algorithm, they must consider how *parallelizable* it is. Figure 2 depicts a parallelizable and non-parallelizable implementation of MPEG-4/H.263 encoder motion estimation. (b) is a parallelizable implementation, (a) is not.

In Figure 2 (a), the motion estimator obtains a guess from the previous macroblock in horizontal scan order [3]. This technique is inherently non-parallelizable due to the fact that the previous macroblock in the chain must be completely processed before the next one can be begun.

Figure 2 (b) shows a different implementation of the adaptive motion estimation algorithm that obtains guess vectors from the corresponding macroblock in the previous frame, rather than the previous macroblock in the current frame. The advantage of this is that the previous frame is already entirely processed, and so the data which is a prerequisite to this macroblock being processed is already available.

IV. Parallelizing an Application

Most approaches to parallelization require two major changes to existing software architecture. These changes are *(i)* compiler extensions and *(ii)* hardware support. There are a few approaches which do not require hardware support [2].

A. Compiler

In order to perform this parallelization the compiler must be extended to support *speculative multithreading*. Harris and Singh [2] have done research on this topic. They have implemented a Haskell compiler called the Glasgow Haskell Compiler (GHC). The GHC parallelizes by operating on *thunks* which are blocks of code allocated by optimized Haskell programs. It attempts to predict which thunks are likely to be good thunks for parallel execution. The ideal thunks are thunks which will be needed by the program and will run long enough that the performance gain will outweigh the overhead introduced.

B. Hardware

Many approaches also require hardware extensions in order to execute their parallelized code. von Praun, Ceze, and Caşcaval [5] are working on one such approach. Their paper ”Implicit Parallelism with Ordered Transactions” presents Implicit Parallelism with Ordered Transactions (IPOT). IPOT is an extension of sequential or explicitly parallel programming models which introduces basic hints to the compiler to say that you want this section of code to be parallelized[4].

IPOT also requires extensions to the hardware to support *speculative multithreading*. The requirements as stated in [5] are as follows:

spawn/commit/squash: Support for creating, committing and squashing speculative tasks. This includes maintaining the correct ordering of speculative tasks as defined by the original sequential program.

conflict detection: Support for the detection of dependence violations. Conflict detection relies on the ordering information and the memory access history of the tasks to determine if there is a violation and which tasks need to be squashed. If a speculative task reads a location that is subsequently written by an ’older’ task, a conflict is flagged and the ’younger’ task is squashed. This

[3]Figure taken from Hwu et al. [3]

[4]Notice the distinction of this from OpenMP where you have to explicitly say not only that you want a section of code parallelized, but also how to parallelize it.

enforcement of dependences guarantees that the original sequential semantics of the program are met.

data versioning: Support for buffering speculative data until commit time. Writes performed speculatively should not be made visible until the task commits. Also related to data versioning is the forwarding of speculative data. Our execution model assumes that speculative versions of data can be provided to more speculative tasks.

V. Related Work

There are a number of proposed implementations with their own distinct strengths and weaknesses Zhong, Lieberman, and Mahlke [6] have proposed one such implementation which makes use of more fine grained parallelization techniques.

VI. Conclusion

Implicit parallelization in general is still a very immature field and there is a great deal of available topics that have yet to be researched. Before implicit parallelization can make it into your every day application, it needs to develop to a point where there are definite performance improvements. Also, researchers need to find an efficient way to decide whether or not to parallelize a particular parallelizable piece of code, dependent on whether or not it will provide a speed increase or be dominated by the overhead introduced.

References

[1] Alberto Bertoni and Marco Dorigo. Implicit parallelism in genetic algorithms. *Artif. Intell.*, 61(2):307–314, 1993. ISSN 0004-3702. doi: http://dx.doi.org/10.1016/0004-3702(93)90071-I.

[2] Tim Harris and Satnam Singh. Feedback directed implicit parallelism. *SIGPLAN Not.*, 42(9):251–264, 2007. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1291220.1291192.

[3] Wen-mei Hwu, Shane Ryoo, Sain-Zee Ueng, John H. Kelm, Isaac Gelado, Sam S. Stone, Robert E. Kidd, Sara S. Baghsorkhi, Aqeel A. Mahesri, Stephanie C. Tsao, Nacho Navarro, Steve S. Lumetta, Matthew I. Frank, and Sanjay J. Patel. Implicitly parallel programming models for thousand-core microprocessors. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 754–759, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1. doi: http://doi.acm.org/10.1145/1278480.1278669.

[4] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D. Owens. Efficient computation of sum-products on gpus through software-managed cache. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 309–318, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3. doi: http://doi.acm.org/10.1145/1375527.1375572.

[5] Christoph von Praun, Luis Ceze, and Calin Caşcaval. Implicit parallelism with ordered transactions. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 79–89, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-602-8. doi: http://doi.acm.org/10.1145/1229428.1229443.

[6] Hongtao Zhong, Steven A. Lieberman, and Scott A. Mahlke. Extending multicore architectures to exploit hybrid parallelism in single-thread applications. In *HPCA '07: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 25–36, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 1-4244-0804-0. doi: http://dx.doi.org/10.1109/HPCA.2007.346182.