Journal of Advanced Computing Research Vol. 5, Issue 1 (2023) 1-5

Parallel Processing - A Case Study on Automatic Parallelization

Abdul Muiz Abdul Ghafar [#1], Mohamed Faidz Mohamed Said [#2]

[#] *Faculty of Computer & Mathematical Sciences, Universiti Teknologi MARA*
70300 Seremban, Negeri Sembilan, MALAYSIA

[1] muizghafar123@gmail.com

[2] mohdfaidz@uitm.edu.my

***Abstract*—Something is parallel if there is an exact level of independence in the order of processes. In other words, it does not matter in what order those processes are executed. Automatic parallelization is an automatic transformation of serial programs into equivalent programs by a compiler. Target may be a multi-core processor namely concurrentization, a vector processor namely vectorization, or a cluster of loosely coupled distributed memory processors namely parallelization. Parallelism mining progression is usually a conversion of source-to-source. It necessitates dependence analysis to identify the dependence between codes. Application of available parallelism is also a challenge. The iterations of a 2-nested loop could be investigated whether they could all be run in parallel. In this paper, a relative study of present and past methods for automatic parallelization is presented. It comprises of methods like array analysis, commutativity analysis, scalar analysis and other similar techniques. The motive of this paper is to provide basic understanding of the methods of automatic parallelization and how these methods are currently being used to generate compilers that automatically develop parallelized applications. Furthermore, the challenges confronted by automatic parallelization are also debated and presented.**

Keywords: **parallelism, automatic parallelization, compilers**

I. Introduction

Parallelization is another optimization technique as mentioned by [1] and [2]. The objective is to lessen the execution time. To this end, multiple processors, or cores are utilized. Automatic parallelization can be defined as transforming serial code into vectorized or multi-threaded code in order to use several processors concurrently in a shared-memory multiprocessor (SMP) machine. The automatic parallelization objective of is to release programmers from the error-prone and complex parallelization process which is executed manually. The need for complex program during compilation remains a grand challenge to achieve fully automatic parallelization of sequential programs.

As a whole, almost all of the processing time of a program are executed inside some form of loop. Therefore, loops are the program control structures where auto parallelization places the most attention. Parallelization of loops consists of two main approaches: cyclic multi-threading and pipelined multi-threading. A cyclic multi-threading parallelizing compiler will separate a loop so that each iteration can be processed on a different processor simultaneously while a pipelined multi-threading parallelizing compiler tries to split up the serial of operations inside a loop to a sequence of code blocks where each block of code can be performed on different processors simultaneously.

II. Literature Review

In this section, the approaches to automatic parallelization are explained. There are many techniques that can be applied in order to optimize automatic parallelization. The techniques stated below are retrieved from several journals.

A. Static and Dynamic Parallelization

According to Mehrara [3], there are two types of automatic parallelization which are static parallelization and dynamic parallelization. In paper [4], the combination of static and dynamic parallelization has also been implemented.

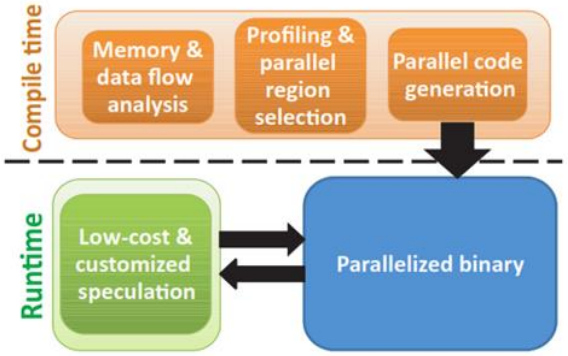

Fig. 1. Static parallelization framework [3]

1) *Static Parallelization:* As shown in Fig. 1, these techniques usually make use of a combination of memory analysis, data flow analysis and profiling to identify potential loops for parallelization and generate parallel binary code during compile time. This parallelized binary is later executed along with a runtime speculation engine and then roll-back in case of any misspeculations.

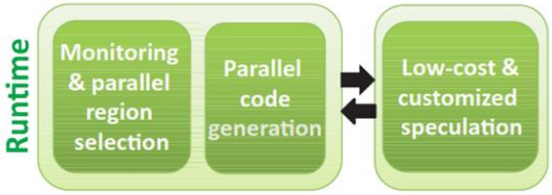

Fig. 2. Dynamic parallelization framework [3]

2) Dynamic Parallelization: As shown in Fig. 2, all steps of the parallelization process including monitoring, parallel region selection and parallel code generation need to be done at runtime, in addition to the runtime speculation.

B. Generalized and Interprocedural Transformations

Author [5] has come out with new approaches to automatic parallelization which are generalized transformations and interprocedural transformations for complete applications.

1) Generalized Transformations: Most earlier work cannot apply parallelization when loops contain conditional control flow. Therefore, a wide selection of loop conversions is improved to deal with conditional branches using the control dependence representation. Bastoul [6] has also been using the same approach as McKinley [5] while authors [8] applied this similar technique in their research but they named it as Generalized Induction Variables (GIV).

2) Interprocedural Transformations: Author [5] introduce two new interprocedural conversions which are loop embedding and loop extraction that expose loop nest to other optimizations without incurring costs related with procedure inlining. The author also presents strategy for determining the benefits and safety of these two transformations when combined with other loop-based optimizations. Authors in [7] also applied this approach in automatic parallelization. Moreover, this approach also has been utilized by authors [8] but they are using different name which is Interprocedural Symbolic Analysis.

C. Scalar and Array Analysis

According to [9], program dependences have to be analysed and observed with every conversion. The full dependence relation is a transitive relation. Direct dependences are well-defined as such dependences that cannot be displayed transitively by other dependences. References [7] and [10] explain that scalar analysis separates a program to analyse the use of scalar variable and to identify the dependencies between these variables. The scalar and array analyses in static single assignment form also have been implemented by [4]. Such cases that can be parallelized due to these dependence problems will be detected by scalar analysis. The segments which are unable to be detected as parallelizable will then be parallelized by array analysis. Furthermore, parallelization can be identified by scalar analysis if it may be allowed or not by using reduction or privatization conversions. Scalar analysis is also utilized to determine dependencies on array elements by their indices. Fig. 3 below shows an example of loop by array analysis.

```
for (i = 0; i < x.length; ++i) {
  for (j = 0; j < i; ++j) {
    x[i * x.length + j] = z[j];
  }
}
```

Fig. 3. Loop by array analysis [6]

Array analysis is the matching part of scalar analysis. Array analysis has one method that is useful on array data to find privatizable arrays. A technique called privatization will assign a duplicate of the working or complete part of the array to each equivalent case that references it as the information that brings zero dependencies to the section in question. Fig. 4 displays a section of code that cannot be privatized even it has a data dependence. The loop needs a conversion of the information in order to parallelize the code section. The array analysis will fail to parallelize this section of program if a conversion cannot be applied.

```
for (i = 0; i < x.length; ++i) {
  for (j = 1; j < x[i].length;
++j) {
    x[i][j - 1] = f (x[i][j]);
  }
}
```

Fig. 4. Non-privatizable array [6]

Reference [8] also described that data can be privatized to loop iteration if the data are used temporarily within the loop so that each processor contributing in the loop execution has different storage for the data. This resolves many data dependences that would rise if all loop repetitions used the same temporary storage for their operations. Very powerful tools provided by these two types of analyses can parallelize code based on scalar and array variables in the loop sections of the program.

Nevertheless, the processor must be able to optimize inter-procedurally in order to exploit the potential of these forms of analysis, thus permitting parallel to extent across function limitations. In an imperative language like C, this analysis is very powerful. Yet, these analytical techniques might not do the trick with a richer language like C++ or Java.

D. Commutativity Analysis

Based on [7] and [10], all the processes where the commutativity analysis will be determined must be separated into object section and an invocation section. Invocation section makes calls to operations while object section provides any access into the receiver. In this segment, the receiver is not available. The compiler uses two conditions in order to test the commutability of operation according to [7].

First, the fresh meaning of individual variable of objects receiver of M and N have to be the similar once the object segment performance of M trailed by the object segment of N as after the object segment performance of N trailed by the object segment of M.

Second, the multiple set of process immediately raised by either M or N below the performance order M followed by N have to be the similar as the multiple set of processes immediately raised by either M or N under the performance order N followed by M.

```
class Node {
  private:
    bool marked;
    int value, sum;
    Node *left, *right;
  public:
    void visit (int);
};

void Node::visit (int p) {
  sum = sum + p;
  if (!marked) {
    marked = true;
    if (left != NULL)
      left->visit (value);
    if (right != NULL)
      right->visit (value);
    }
}
```

Fig. 5. Example of commutativity analysis [6]

Fig. 5 taken from the journal by [7] shows a code section that can be identified to be commutative. The method that has the object as a receiver retrieves only the instance variables. The language become a bit more tough to program in due to these limitations.

```
void Calculator::calculate
(Stack s) {
  if (!s.empty) {
    value = this->operate (s,
value);
  }
}
```

Fig. 6. Commutativity analysis cannot parallelize this method [6]

Fig. 6 demonstrates a section of source code where the commutativity analysis cannot be parallelized. Other than that, the instance variable is assigned with a value that is coded in a way that is not divisible. The code:

```
value = this->operate (s, value)
```

denies the separability rules. The object segment is entwined with the invocation part. To do its calculations, the object segment relies on the invocation segment. This can be a crucial problem in today's programming atmosphere.

III. METHODOLOGY

A. High Level Parallelization

As stated by [10], high level parallelization is known as a method for putting parallelized output into an intermediate language. The main focus of this method is on the construction of a parallel run-time library that is to which parallelized codes are linked. There are three stages in high level parallelization according to [7].

1) Scalarization: This technique transforms specific array programs of FORTRAN 90 into corresponding loops while protecting the semantic of the programs.

2) Transformation: This conversion stage optimizes loops for single processor machines.

3) Interprocedural Analysis and Inlining: The parallelization optimizations is concentrated in this phase. To improve the parallelization results, an interprocedural data flow analysis is executed. But presently, the segment of interprocedural data flow has been removed from the processor.

This method has been enhanced, with improvement such as outlining, select iteration reordering transformations and locality optimizations. The second improvement was outlining, a technique that can be relatively defined as the contradict of inlining. The procedure contains regions definition of the statement and statement combination of a procedure.

B. Instruction-Level Parallelism

Parallel computers, as well as research on parallel languages, compilers, and distribution techniques, first emerged in the late 1960s. Bliss [11] states that instruction-level parallelism has been one of the most successful research areas, which refers to the effort of determining instructions in the program that can be performed out of order or in parallel and arrange them to decrease the computation time.

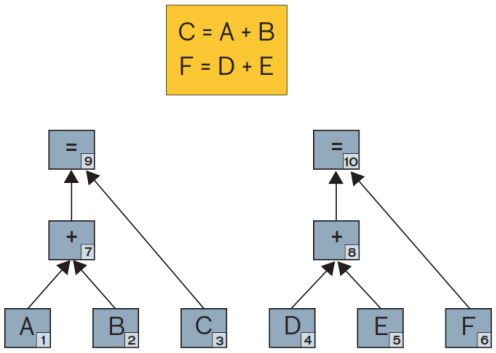

Fig. 7. A program is represented by a parse tree or signal flow graph [7]

Fig. 7 illustrates a simple program and an associated signal flow graph or parse tree. Note that there are no dependencies between the computation of C and F. Nodes 4, 5, 6, 8, and 10 are not connected to nodes 1, 2, 3, 7, and 9. The two addition operations can be executed in parallel since they are lack of dependencies. If the structure allows for several instructions to be executed at once, this method can significantly speed up program performance. Such instruction-level parallelism has been combined into a number of mainstream compilers.

Unfortunately, instruction-level parallelism does not solve the automatic parallelization completely. It is not enough to build a signal flow graph or parse tree of a program, identify what nodes can be performed in parallel and then split up the program accordingly.

C. Parallel Reductions

This approach was applied by reference [8]. The program of the type sum = sum + a(i) where i is the loop index show a reappearance pattern that generally must be executed in sequence. However, a parallel execution is possible because the sum execution is mathematically commutative and associative. This can be done by collecting partial sums on each processor, and then summing the partial results, as shown in Fig. 8. The partial results may be summed later when the loop in a critical segment. Note that this conversion may change the output because reordering the sum processes may bring to round-off errors that are different from those in the original code. The codes in the Perfect Benchmarks suite have not been found to be sensitive to such reorganization.

```
REAL a(m)
DO i=1,n
  ...
  expr = ...
  a(t(i)) = a(t(i)) + expr
ENDDO
```

↓

```
REAL a(m),a1(m, number_of_processors)
CDOALL i=1,m
  a1(i,1:number_of_processors) = 0
ENDDO
CDOALL i=1,n
  ...
  expr = ...
  a1(t(i),my_proc) = a1(t(i),my_proc) + expr
ENDDO
CDOALL i=1,m
    a(i)=a(i)+SUM(a1(i,1:number_of_processors))
ENDDO
```

Fig. 8. Expanded parallel reduction transformation [8]

D. Parallel Loops Mapping

This technique has been tested by authors [8] by using the restructuring compiler as a starting point for hand-optimized codes to discover parallelism, but then it is mapped to the machine poorly. They use various methods to improve this technique such as strip-mining, loop coalescing, outer loop parallelization, loop fusion and data localization.

E. Polyhedral Transformation Framework

The task of program optimization for parallelism and locality in the polyhedral model may be viewed in terms of three phases according to reference [12]. Those phases are static dependence analysis of the input program, transformations in the polyhedral abstraction and generation of code for the transformed program. Authors [12] have prototyped an end-to-end practical parallelizer and locality optimizer in the polyhedral model. Their system generates tiled code for statement domains of arbitrary dimensionalities under statement-wise affine transformations for data locality optimization as well as shared memory parallel execution. Fig. 9 exhibits the components of their prototype system for automatic parallelization. Other than this paper [12], this approach has also been implemented on Intel Many Integrated Core (IMIC) by authors [13].

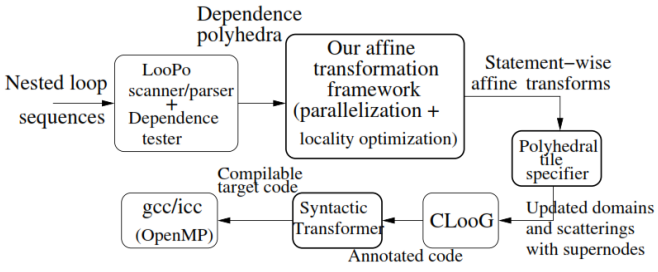

Fig. 9. Source-to-source transformation system prototype [9]

F. AutoFutures: Automatic Asynchronous Method Calls

In reference [14], AutoFutures was presented, a method that automatically determines parallelizable parts in sequential code and then reorganize them for multicore. This method stresses minimal alteration to sequential code. AutoFutures increase the acceptance of parallel software and make parallel code easy to understand. Static analysis was used to determine code that can be processed asynchronously.

Without any data dependencies, these precondition problems will search space for parallelization possibility down to code that can verifiably be processed in parallel. This could be part of a processor. The synchronization has to be addressed after code hotspots have been determined. A placeholder for the result of an asynchronous computation is being served by a AutoFutures. AutoFutures propose an easy way to hide synchronization code and specify asynchronicity.

G. Generalized and Interprocedural Transformations

Jablin [15] on the other hand, has applied automatic parallelization in Graphics Processing Unit (GPUs) by using pipeline parallelism techniques. Pipeline parallelism extends the applicability of GPUs by exposing independent work units for code with loop-carried dependences. A pipeline consists of several stages distributed over multiple threads. Each stage executes in parallel with data passing from earlier to later stages through high-speed queues. Automatic pipeline parallelization techniques construct pipelines from sequential loops by partitioning instructions into different stages [16]. Careful partitioning segregates dependent and independent operations. Stages with loop-carried dependences are called sequential stages. Stages without loop-carried dependences are called parallel stages. Each iteration of a parallel stage can execute independently on different processors.

IV. Conclusion

As computer technologies are expanding rapidly in the world of computing, auto parallelization in a processor is becoming more significant. Automatic parallelization has become a crucial step in developing well-organized code for various multithreading applications. In this paper various methods of achieving automatic parallelization has been discussed. It is found that scalar and array analysis when used in combination act as a powerful tool for parallelization of applications. By using the subset of C++, commutativity analysis worked so well. At present, there are multiple techniques combination of parallelization in a compiler.

Many obstacles occur in the exhibition of automatic parallelization for large-scale computational applications. These problems are required to be resolved when automatic parallelization has to be demonstrated on large-scale computational applications. It is clear that parallelization is not fully automatic yet. There are obvious methods that need to be followed to allow compilers with automatic parallelization. There are many tools for demonstrating automatic parallelization. Though the parallel codes can be developed by automatic parallelization tools, additional attempts are needed to optimize those codes in matters of performance. These tools should make an effort to omit the loops with smaller execution time.

REFERENCES

[1] Y. N. Srikant, *Automatic Parallelization - Part 1*. Bangalore: Department of Computer Science, Indian Institute of Science, 2011, pp. 1-6.

[2] V. D. P. Ruud, *Basic Concepts in Parallelization*. California: Oracle Solaris Studio, 2010, pp. 1-6.

[3] M. Mehrara, "Static and Dynamic Parallelization: Challenges and Opportunities", *Compiler and Runtime Techniques for Automatic Parallelization of Sequential Applications*, pp. 5-8, 2011.

[4] D. R. Chakrabarti and P. Banerjee, "Global Optimization Techniques for Automatic Parallelization of Hybrid Applications", *Proceedings of the 15th International Conference on Supercomputing*, pp. 166-180, 2001.

[5] K. S. McKinley, "Automatic Parallelization", *Automatic and Interactive Parallelization*, pp. 12-32, 1994.

[6] C. Bastoul, "Efficient Code Generation for Automatic Parallelization and Optimization", *Proceedings of the Second International Conference on Parallel and Distributed Computing*, 2003.

[7] N. DiPasquale, V. Gehlot, and T. Way, "Comparative Survey of Approaches to Automatic Parallelization", *MASPLAS'05*, pp. 1-6, 2005.

[8] K. Molitorisz, J. Schimmel, and F. Otto, "Automatic Parallelization using AutoFutures", *Multicore Software Engineering, Performance, and Tools*, 2012.

[9] T. Brandes, "The Importance of Direct Dependences for Automatic Parallelization", *Proceedings of the 2nd International Conference on Supercomputing*, pp. 408-417, 2012.

[10] M. Sohal and R. Kaur, "Automatic Parallelization: A Review", *International Journal of Computer Science and Mobile Computing*, vol. 5, no. 5, pp. 17-21, 2016.

[11] N. Bliss, "Addressing the Multicore Trend with Automatic Parallelization", *Lincoln Laboratory Journal*, vol. 17, pp. 187-198, 2007.

[12] R. Eigenmann, D. Padua, and J. Hoeflinger, "New Transformation Techniques: Performance and Relevant Code Patterns", *On the Automatic Parallelization of the Perfect Benchmarks*, pp. 4-20, 1998.

[13] K. Stock, L. N. Pouchet, and P. Sadayappan, "Automatic Transformations for Effective Parallel Execution on Intel Many Integrated Core", *TACC-Intel Highly Parallel Computing Symp*, pp. 1-6, 2001.

[14] U. Bondhugula, M. Baskaran, A. Hartono, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Towards Effective Automatic Parallelization for Multicore Systems", *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium*, 2008.

[15] T. B. Jablin, "Pipeline Parallelism", *Automatic Parallelization for GPUs*, pp. 10-13, 2013.

[16] A. Muiz, "171129 CSC580 C1 AMAG Youtube", 2017. [Online]. Available:https://www.youtube.com/watch?v=sG0JLcH9Nv4. [Accessed: 10-Dec-2017].