Automatic Parallelization Tools

Ying Qian

***Abstract*—In recent years, much has been made of the computing industry's widespread shift to parallel computing. Nearly all consumer computers will ship with multicore processors. Parallel computing will no longer be only relegated to exotic supercomputers or mainframes, moreover, electronic devices such as mobile phones and other portable devices have begun to incorporate parallel computing capabilities. High performance computing becomes increasingly important. To maintain high quality solutions, programmers have to efficiently parallelize and map their algorithms. This task is far from trivial, especially for different existing parallel computer architectures and different parallel programming paradigms. To reduce the burden on programmers, automatic parallelization tools are introduced. The purpose of this paper is to discuss different automatic parallelization tools for different parallel architectures.**

***Index Terms*—parallel processing, automatic parallelization tools, parallel programming paradigms**

I. Introduction

THE parallel machines are being built to satisfy the increasing demand of higher performance for parallel applications. Multi and many-core architectures are becoming a hot topic in the fields of computer architecture and high-performance computing. Processor design is no longer driven by high clock frequencies. Instead, more and more programmable cores are becoming available. In recent years, much has been made of the computing industry's widespread shift to parallel computing. Nearly all consumer computers will ship with multicore processors. Parallel computing will no longer be only relegated to exotic supercomputers or mainframes, moreover, electronic devices such as mobile phones and other portable devices have begun to incorporate parallel computing capabilities.

Therefore, more and more software developers will need to cope with a variety of parallel computing platforms and technologies in order to provide optimum performance to fully utilize all the processor power. However, the parallel programming requires in-depth knowledge on underlying systems and parallel paradigms, which make the whole process difficult. It would be desirable to let programmer program in sequential and have the parallelization tools help to automatic parallelize the program.

Depending on different parallel computer architectures or machines, the parallel applications can be written using a variety of parallel programming paradigms, including message passing, shared memory, data parallel, bulk synchronous, mixed-mode and so on. The message passing and shared memory paradigms are the two most important programming paradigms.

The general procedure to parallelize an application can be concluded as, 1) decide on which kind of parallel computer system will be used, 2) choose the right parallel programming paradigm to parallelize the code, 3) choose the good automatic parallelization tool.

The rest of paper is organized as follows. In Section 2, I provide the background introduction of several popular parallel computer architectures. The parallel programming paradigms which are designed for these architectures are discussed in Section 3, which include shared memory, message passing programming paradigm and so on. Section 4 introduces the automatic parallelization and general way to employ it. Finally the existing automatic parallelization tools are discussed in Section 5.

Manuscript received July 15, 2012; revised August 9, 2012. This work was supported by KAUST Supercomputing Laboratory at King Abdullah University of Science and Technology (KAUST).

Y. Qian is with KAUST Supercomputing Laboratory, King Abdullah University of Science and Technology, Thuwal, 23955-6900, Saudi Arabia (e-mail: ying.qian@kaust.edu.sa).

II. Parallel computer architectures

In the past decade, high performance computers have been implemented using a variety of architectures. I briefly describe the most common parallel computer architectures here.

Based on the Flynn's classification [1], there are four kinds of machine architectures, single-instruction stream single-data stream (SISD), single-instruction stream multiple-data streams (SIMD), multiple-instruction streams single-data stream (MISD) and multiple-instruction streams multiple-data streams (MIMD). SISD models conventional sequential computers. MISD was seldom used. In an SIMD machine, all processors execute the same instruction at the same time. It is a synchronous machine, and mostly used for special purpose applications. An MIMD machine is a general-purpose machine, where processors operate in parallel but asynchronously. MIMD machines are generally classified into four practical machine models: Symmetric Multiprocessors (SMP), Massively Parallel Processors (MPP), Distributed Shared Memory (DSM) multiprocessors, Cluster of Workstations (COW), and Cluster of Multiprocessors (CLUMP).

A. SMP

SMP is a Uniform Memory Access (UMA) system, where all memory locations are the same distance away from the processors, so it takes roughly the same amount of time to access any memory location. SMP systems have gained prominence in the market place. Considerable work has gone into the design of SMP systems, and several vendors such as IBM, Compaq, SGI, and HP offer small to large-

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

scale shared memory systems [2]. A typical SMP machine architecture is shown in Fig. 1.

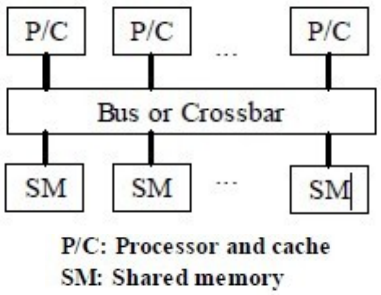

Fig. 1. A typical SMP machine.

B. Multi-core Cluster

Multi-core clusters are distributed-memory systems, where there are multiple nodes each having multiple processors and its own local memory. For these systems, one node's local memory is considered remote memory for other nodes. SMPs are called tightly coupled [1]. 81.4% of the top 500 supercomputers in the world are clusters [3]. A typical multi-core cluster machine is shown in Fig. 2.

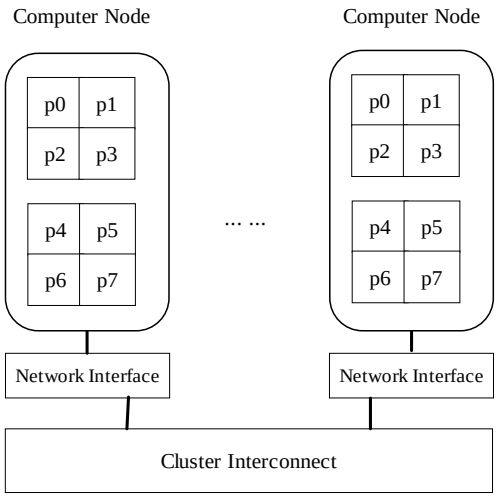

Fig. 2. A typical multi-core cluster.

C. GPU

At the same time, Graphics Processing Units (GPUs) are becoming programmable. While their graphics pipeline was completely fixed a few years ago, now, we are able to program all main processing elements using C-like languages, such as CUDA [4] or OpenGL [5]. Multi-core and many-core architectures are emerging and becoming a commodity. Many experts believe heterogeneous processor platforms including both GPUs and CPUs will be the future trend [6]. A typical GPU architecture as in Fig. 3 consists of a number of Single Instruction Multiple Thread (SIMT) processor clusters. Each cluster has access to an on-chip shared memory (*smem*), a L2 cache and a large off-chip memory.

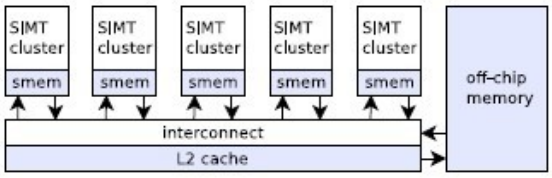

Fig. 3. Schematic view of a GPU architecture. [6]

III. PARALLEL PARADIGMS

Parallel computers provide support for a wide range of parallel programming paradigms. The HPC programmer has several choices for the parallel programming paradigm, including the shared memory, message passing, mix-mode, and GPU.

A. Message passing

MPI [7] is a well-known message passing environment. MPI has good portability, because programs written using MPI can run on distributed-memory systems, shared-memory multiprocessors, and networks of workstations or clusters. On top of shared memory systems, message passing is implemented as writing to and reading from the shared memory. So MPI can be implemented very efficiently on top of the shared memory systems. Another advantage of the MPI programming model is that the user has complete control over data distribution and process synchronization, which can provide optimal data locality and workflow distribution. The disadvantage is that existing sequential applications require a fair amount of restructuring for parallelization based on MPI.

MPI provides the user with a programming model where processes communicate with each other by calling library routines.

B. Shared Memory

Message-passing codes written in MPI are obviously portable and should transfer easily to SMP cluster systems. However, it is not immediately clear that message passing is the most efficient parallelization technique within an SMP box, where in theory a shared memory model such as OpenMP [8] should be preferable. OpenMP is a loop level programming style. It is popular because it is easy to use and enables incremental development. Parallelizing a code includes two steps, (1) discover the parallel loop nests contributing significantly to the computations time; (2) add directives for starting/closing parallel regions, managing the parallel threads (workload distribution, synchronization), and managing the data.

C. Mix-mode

In the mixed MPI-OpenMP programming style, each SMP node executes one MPI process that has multiple OpenMP threads. This kind of hybrid parallelization might be beneficial when it utilizes the high optimization of the shared memory model on each node. As small to large SMP clusters become more prominent, it is open to debate whether pure message-passing or mixed MPI-OpenMP is the programming of choice for higher performance.

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

D. CUDA

In comparison to the central processor's traditional data processing pipeline, performing general-purpose computations on a graphics processing unit (GPU) is a new concept. CUDA [4] are specifically designed for general purpose GPU programming. The CUDA architecture included a unified shader pipeline, allowing each and every arithmetic logic unit (ALU) on the chip to be marshaled by a program intending to perform general-purpose computations. These ALUs were built to comply with IEEE requirements for single-precision floating-point arithmetic and were designed to use an instruction set tailored for general computation rather than specifically for graphics. Also, the execution units on the GPU were allowed arbitrary read and write access to the memory and software managed cache known as shared memory.

Even so, the mapping process of most applications remains non-trivial. To achieve this, the programmer has to deal with mappings of distributed memories, caches, and register files. Additionally, the programmer is exposed to parallel computing problems such as data-dependencies, race-conditions, synchronization barriers and atomic operations.

IV. Automatic parallelization

A. Need for automatic parallelization

High performance computing becomes increasingly common. To fully utilize these power machines, programmers have to efficiently parallelize and map their applications. This task is far from trivial, leading to the necessity to automate this process.

Past techniques provided solution for languages like FORTRAN and C; however, these are not enough. These techniques dealt with parallelization sections with specific systems in mind like loop or particular section of code. Identifying possibilities for parallelization is a crucial step for generating parallel application. This need to parallelize applications is partially addressed by tools that analyze code to exploit parallelism. These tools use either compile time techniques or run-time techniques. Some techniques are built-in in some parallelizing compilers but user needs to identify parallelize code and mark the code with special language constructs. The compiler identifies these language constructs and analyzes the marked code for parallelization. Some tools only parallelize special form of code like loops. Hence a fully automatic tool for converting sequential code to parallel code is still required.

B. General procedure of parallelization

The process starts with identifying code sections that the programmer feels have parallelism possibilities. Often this task is difficult since the programmer who wants to parallelize the code has not originally written the code. Another possibility is that the programmer is new to the application domain. Therefore, this first stage in the parallelization process seems easy at first, but it may not be straightforward.

The next stage is to identify the data dependency relation of given sequential program. This is a crucial stage to sort code sections out of the identified ones that actually need parallelization. This stage is the most important and difficult since it involves lot of analysis.

Most research compilers for automatic parallelization consider Fortran programs, because during this data dependency identification stage, Fortran makes stronger guarantees about aliasing than languages such as C and C++. Aliasing can be described as a situation where a data location in memory can be accessed through different symbolic names in the program. Thus, modifying the data through one name implicitly modifies the values associated to all aliased names, which may not be expected by the programmer. As a result, aliasing makes it particularly difficult to decide the data dependency of programs. Aliasing analyzers intend to make and compute useful information for understanding aliasing in programs. Therefore, generally for codes in C/C++ where pointers are involved are difficult to analyze. If there are more dependencies in the identified code sections, the possibilities of parallelization decreases.

Sometimes the dependencies are removed by changing the code and this is the next stage in parallelization. Code is transformed such that the functionality and hence the output is not changed but the dependency, if any, on other code section or other instruction is removed.

The last stage is to generate the parallel code with appropriate parallel programming model. Functionally this code should be same to the original sequential code. Moreover, it has additional constructs and parallel function calls to enable it to run on multiple threads, processes or both.

V. Parallelization tools

A. OPENMP

ICU_PFC

ICU_PFC is introduced in [9], which is an automatic parallelizing compiler. It receives FORTRAN sources and generates parallel FORTRAN codes where OpenMP directives for parallel execution are inserted. ICU-PFC detects the DO ALL parallel loop in the program, and inserts appropriate OpenMP directives. For parallel loop detection, a dependency matrix is designed to store data dependency information of statements in a loop. Finally, the parallelized code generator of ICU-PFC can generate OpenMP supported parallel code.

Polaris compiler

The Polaris compiler [10] takes a Fortran77 program as input, transforms this program so that it runs efficiently on a parallel computer, and outputs this program version in one of several possible parallel FORTRAN dialects. Polaris performs its transformations in several "compilation passes". In addition to many commonly known passes, Polaris includes advanced capabilities performing the following tasks: array privatization, data dependence testing, variable recognition, inter procedural analysis, and symbolic program analysis.

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

Cetus

The Cetus [11] tool provides an infrastructure for research on multicore compiler optimizations that emphasizes automatic parallelization. The compiler infrastructure, which targets C programs, supports source-to-source transformations, is user oriented and easy to handle, and provides the most important parallelization passes as well as the underlying enabling techniques.

The infrastructure project follows Polaris [10], which is a research infrastructure for optimizing compilers on parallel machines. While Polaris translated Fortran, Cetus targets C programs.

SUIF

The SUIF [12] (Stanford University Intermediate Format) compiler is an automatic parallelizing compiler made at Stanford university. This compiler reads ordinary C or FORTRAN programs and generates parallelized intermediate codes (of the Stanford University Intermediate Format), parallelized executable codes for some systems, or C program containing parallel constructs such as doall, doacross, etc. It is able to detect and generate doalls, with freely available package. This compiler is composed of many passes. Each pass does its role and is independent of each other.

The freely available SUIF compiler finds every parallelism for all the loops contained in a given source program and makes an effort to maximize the granularity and to minimize communications and synchronizations as much as it can. And it generates intermediate codes containing tags indicating "parallelizable". Using the intermediate codes, the pass *pgen* generates a parallelized intermediate code for shared memory multiprocessors.

Intel Compiler

The auto-parallelization feature of the Intel Compiler [13] automatically translates serial portions of the input program into semantically equivalent multi-threaded code. C, Fortran and C++ are all supported. The auto-parallelization feature is only for SMP type of machine.

Automatic parallelization determines the loops that are good candidates for parallelization. Then, it performs the data-flow analysis to verify correct parallel execution, and partitions the data for threaded code generation, and inserts with OpenMP directives properly.

B. MPI

Automatic parallelizing compilers such as SUIF [12] and Polaris [10] can be a solution for SMP machines. For distributed memory parallel processor systems or heterogeneous computers, message passing paradigm is usually used. For such kind of machines, a compiler backend must convert a shared memory based parallel program into a program using message passing scheme, send/receive.

Based on the automatic parallelizing compiler SUIF, a new parallelization tool is introduced in [14]. The original backend of SUIF is substituted with a new backend using MPI, which give it the capability of validating of parallelization decisions based on overhead parameters.

This backend converts shared-memory based parallel program into distributed-memory based parallel program with MPI function calls without excessive parallelization that causes performance degradation.

The tool adopts the SUIF parallelizing compiler and the LAM implementation [15] of MPI for the patchwork system. The tool converts the implicit data sharing and explicit doall into explicit data send/receive functions and implicit doall into explicit descriptions of condition for the sequential execution of the master node. Eventually it converts a program assuming a shared memory environment into one assuming the other.

In a shared memory environment, all data in memory can be accessed by every processor. In a distributed environment, all data must be sent and received from the processing node explicitly which has the data, to the nodes which want to use them. Therefore, the remote data must be identified. It is done by using the dataflow equation [16] and DEF and USE sets provided by SUIF, for every doall loop represented in SUIF. There is no need to examine all the code blocks for such data, since SUIF parallelize loops at current state. The tool incorporates function calls for sending and receiving messages into the parallel code generator of SUIF, since MPI uses the traditional send/receive message scheme. After completion of the doall loop, the result should be returned to the master.

The code generated by SUIF contains no explicit master node but it is started on a node and the node calls the doall function, with the function pointer that points to the function containing the actual doall loop, whenever doall is reached. The parallel code generator *pgen* is modified to insert a conditional statement that makes the current processor execute a block that is not parallelized, if the node id of the current processor is the same as that of the master. By these, the sequential part of the program assuming shared memory could be converted into one assuming distributed memory.

C. GPU

In [17], a technique to automatically parallelize and map sequential image processing algorithm on a GPU is presented. The work is based on *skeletonization*, which separates the structure of a parallel computation from the algorithms functionality, enabling efficient implementations without requiring architecture knowledge from the programmer. [17] A number of skeleton classes are defined for image processing algorithms. Each skeleton class enables specific GPU parallelization and optimization techniques, such as automatic thread creation, on-chip memory usage and memory coalescing.

The tool uses domain specific skeletons and a fine grained classification of algorithms. If we compare skeleton-based parallelization to existing GPU code generators in general, skeleton-based parallelization potentially achieves higher hardware efficiency by enabling algorithm restructuring through skeletons.

D. EMBEDDED SYSTEMS

The OSCAR [18] automatic parallelizing compiler has

ISBN: 978-988-19251-6-9
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

been developed to utilize multicores for consumer electronics like portable electronics, mobile phones, car navigation systems, digital TVs and games. Also, a new Consumer Electronics Multicore Application Program Interface (API) is defined [19] to use the OSCAR compiler with native sequential compilers for various kinds of multicores from different vendors. It has been developed in NEDO (New Energy and Industrial Technology Development Organization) "Multicore Technology for Realtime Consumer Electronics" project with six Japanese IT companies.

To parallelize a program on consumer electronics multicores, it is especially important to optimize memory allocations, overlap data transfer and realize low power consumption.

This API uses a subset of OpenMP. Furthermore, this API enables the allocation of various kinds of memory such as local memory, distributed shared memory and on-chip central shared memory for real-time processing. It also realizes low power consumption on multicore memory architectures. The automatic parallelizing compiler generates parallelized programs with the API, so that the generated parallel programs can be easily executed on different processor cores and memory architectures.

VI. Conclusion

High performance computing becomes increasingly important. To maintain high quality solutions, programmers have to efficiently parallelize and map their algorithms. This task is far from trivial, leading to the necessity to automate this process. The general procedure to parallelize an application can be concluded as, 1) decide on which kind of parallel computer system will be used, 2) choose the right parallel programming paradigm to parallelize the code, 3) choose the good automatic parallelization tool.

A number of automatic penalization tools are discussed in this paper. Most of the tools are designed for SMP, shared memory machines, where OpenMP is supported. I believe OpenMP kind of parallel codes are easier to be parallelized than other parallel paradigms, such as MPI. Few works have been done to automatically parallelize MPI, CUDA codes and also embedded systems, such as portable multi-core devices. However in real life, the high scalable systems are mainly clusters, which are distributed memory systems. We really need automatic parallelization tools for MPI or MPI/OpenMP mix-mode applications.

Acknowledgment

I would like to thank Dr. Kai Qian for his kind support.

References

[1] K. Huang, Z. Xu, "Scalable Parallel Computing: Technology, Architecture, Programming".
[2] N. R. Fredrickson, Ahmad Afsahi, and Ying Qian, "Performance Characteristics of OpenMP Constructs, and Application Benchmarks on a Large Symmetric Multiprocessor," *17th Annual ACM International Conference on Supercomputing, ICS'03, San Francisco, CA, USA, June, 2003, pp. 140-149.*
[3] Top500 supercomputer sites. Available: http://www.top500.org
[4] CUDA. Available: http://www.nvidia.com/object/cuda_home_new.html
[5] OpenGL. Available: http://www.opengl.org
[6] S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A Compiler Framework for Automatic Translation and Optimization," *Proc. ACM Symp. Principles and Practice of Parallel Programming (PPoPP 09), ACM Press, Feb. 2009, pp. 101-110.*
[7] *Message Passing Interface Forum: MPI, A Message Passing Interface Standard, Version 1.2, 1997.*
[8] *OpenMP C/C++ Application Programming Interface, Version 2.0, March 2002.*
[9] H. Kim, Y. Yoon, S. Na, D. Han, "ICU-PFC: An Automatic Parallelizing Compiler," *International Conference on High Performance Computing in the Asia-Pacific Region - HPCASIA , 2000.*
[10] D. A. Padua, R. Eigenmann, J. Hoeflinger, P. Petersen, P. Tu, S. Weatherford, and K. Faigin, "Polaris: A new-generation parallelizing comiler for MPPs," Technical Report CSRD-1306, Center for Supercomputing Research and Development, Univ. of Illinois at Uurbana-Champaign, June 1993.
[11] C. Dave, H, Bae, S. Min, S. Lee, R. Eigenmann, and S. Midkiff, "Cetus: A Source-to-Source Compiler Infrastructure for Multicores," *Computer, vol. 42, no. 12, pp. 36-42, Dec. 2009.*
[12] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E Bugnion, and M.S. Lam, "Maximizing Multiprocessor Performance with the SUIF compiler," *IEEE Computer, December 1996.*
[13] Intel compiler. Available: http://software.intel.com/en-us/articles/intel-compilers
[14] D. Kwon and S. Han, "MPI Backend for an Automatic Parallelizing Compiler," *1999 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '99), Fremantle, Australia , June 1999.*
[15] MPI Primer / Developing with LAM (manual), Ohio Supercomputer Center, 1996.
[16] K. Li, "Predicting the Performance of Partitionable Multiprocessors," *Proc. Of PDPTA'96 International Conference, pp 1350-1353., 1996.*
[17] C. Nugteren, H. Corporaal, and B. Mesman, "Skeleton-based Automatic Parallelization of Image Processing Algorithms for GPUs," *ICSAMOS, 2011, pp. 25-32.*
[18] H. Kasahara, M. Obata, K. Ishizaka, K. Kimura, H. Kaminaga, H. Nakano, K. Nagasawa, A. Murai, H. Itagaki, and J. Shirako, "Multigrain Automatic Parallelization in Japanese Millennium Project IT21 Advanced Parallelizing Compiler," *PARELEC IEEE Computer Society (2002) , p. 105-111.*
[19] T. Miyamoto, S. Asaka, H. Mikami, M. Mase, Y. Wada, H. Nakano, K. Kimura, and H. Kasahara, "Parallelization with Automatic Parallelizing Compiler Generating Consumer Electronics Multicore API," *International Symposium on Parallel and Distributed Processing with Applications, 2008. ISPA '08, pp,600-607,*