Advances in Engineering Software 000 (2017) 1–8

Contents lists available at ScienceDirect

Advances in Engineering Software

journal homepage: www.elsevier.com/locate/advengsoft

Type oriented parallel programming for Exascale

Nick Brown

Edinburgh Parallel Computing Centre, James Clerk Maxwell Building, Kings Buildings, Edinburgh, United Kingdom

ARTICLE INFO

Article history:
Received 24 April 2015
Revised 20 March 2017
Accepted 20 April 2017
Available online xxx

Keywords:
Type oriented programming
Mesham
Parallel programming
Type systems
Asynchronous Jacobi
PGAS

ABSTRACT

Whilst there have been great advances in HPC hardware and software in recent years, the languages and models that we use to program these machines have remained much more static. This is not from a lack of effort, but instead by virtue of the fact that the foundation that many programming languages are built on is not sufficient for the level of expressivity required for parallel work. The result is an implicit trade-off between programmability and performance which is made worse due to the fact that, whilst many scientific users are experts within their own fields, they are not HPC experts.

Type oriented programming looks to address this by encoding the complexity of a language via the type system. Most of the language functionality is contained within a loosely coupled type library that can be flexibly used to control many aspects such as parallelism. Due to the high level nature of this approach there is much information available during compilation which can be used for optimisation and, in the absence of type information, the compiler can apply sensible default options thus supporting both the expert programmer and novice alike.

We demonstrate that, at no performance or scalability penalty when running on up to 8196 cores of a Cray XE6 system, codes written in this type oriented manner provide improved programmability. The programmer is able to write simple, implicit parallel, HPC code at a high level and then explicitly tune by adding additional type information if required.

1. Introduction

The difficulty of programming has been a challenge to parallel computing over the past several decades [1] and as the community moves towards Exascale, where it is likely that one will take advantage of more and more cores to solve problems, then this will become more severe. It is critical that end programmers, who might not be HPC experts, can write their code in an abstract yet powerful and consistent manner if they are to take full advantage of future super computers.

Parallel programming models largely fall into two categories: explicit parallelism and implicit parallelism. When using explicit parallelism the programmer must handle all details of data allocation, partition, distribution, communication and synchronization which is notoriously difficult. Even to the few experts explicit parallel programs, such as those using MPI, are low-level and difficult to develop, test, debug and modify. Implicitly parallel languages, which are simpler, rely on more hidden optimization - the compiler essentially makes decisions for the end programmer. However it is not always easy to automatically make the right decisions for parallelism. For example, the initial partition and distribution of an array can significantly affect the performance of later computation and different parts of a code may require the same array to be partitioned in different directions. This is why most parallel codes currently used in real applications are hand-written and explicitly parallel - because hardware and compiler technology is not yet advanced enough to guarantee scalable and performant implicit parallelism.

This paper proposes a trade off between explicit parallelism and implicit parallelism. Type oriented programming addresses the issue by providing the options to the end programmers to choose between explicit and implicit parallelism. The approach is to design new types governing parallelism where a programmer may choose to use these types or may choose not to use them. These types impose additional information that guides the compiler to generate the required parallel code or conduct optimization and apply some default parallelism method when detailed information is missing. In short these types for parallelism are issued by the programmer to instruct the compiler to perform the expected actions in static analysis and code generation. They are predefined by expert HPC programmers in a type library and different target communication methods correspond to different combinations of types.

In [2] we introduced the idea of type oriented programming and this paper focuses on applying these ideas to solve some of the challenges associated with Exascale and studying the performance

E-mail address: nick.brown@ed.ac.uk

http://dx.doi.org/10.1016/j.advengsoft.2017.04.006
0965-9978/

and scaling behaviour at far greater core counts. As one moves towards Exascale, where the programmer must make efficient use of a vast amount of resource, scalability issues will force the common algorithms for solving problems to be reconsidered. The lack of suitable programming model will limit the scientific benefit that one can gain from this new generation of machine. Jacobi's algorithm is an example of a difficulty the community might face - whilst using asynchronous halo swap communication between iterations can help improve both scability and performance at large core counts it also requires far more complex code to be written. In this paper we introduce the type oriented parallel programming language, Mesham, and using this language implement versions of Jacobi's algorithm using synchronous and asynchronous communication methods. We consider the programming benefits of expressing asynchronous Jacobi's algorithm in a type oriented manner and demonstrate performance and scalability of our approach with runs using up to 8196 cores on a Cray XE6.

The rest of the paper is organised as follows: Section 2 reviews the background to the problem and in Section 3 we present our proposed solution. Section 4 introduces the parallel programming language, Mesham, which is used in Section 5 where we consider both the programmability and performance characteristics of our approach when applied to a case study. Section 5 draws some conclusions and considers further work.

2. Background

It is widely accepted that writing parallel codes is far more complex than their sequential counterparts. Factors such as data decomposition, communication and synchronisation add additional complexity that HPC users, who are often not programming experts, can find difficult to handle. Decisions made early on, such as the method of data distribution, might be naive in the benefit of hindsight but can prove very difficult to change once the code has matured. When it comes to writing parallel codes, there is a trade off between languages. Current practice is to write parallel codes using some lower level sequential language such as C or Fortran combined with a communications library such as MPI or OpenMP. This existing approach requires the programmer to construct their code whilst considering many levels of abstraction ranging from the high level parallel system all the way down to complex low level sequential details. Other parallel programming solutions place more emphasis upon simplicity and maintainability; however the programmer can often be stuck with some default choices which impacts performance and scalability. This illustrates the fundamental trade-off in many parallel programming solutions; those solutions which provide detailed control where the programmer can tune every aspect of parallelism to achieve good performance and scalability result in complex, difficult to maintain programs and those languages which abstract the programmer sufficiently to promote simplicity but at the expense of scalability and performance.

As we move towards Exascale, and one harnesses more and more processors to solve a problem, this challenge is likely to become even more acute. It will be very difficult to use those existing languages which allow for high levels of control, and those that make decisions and abstraction in the name of simplicity will likely scale poorly.

2.1. Type oriented programming

A large subset of languages follow the syntax *Type Variablename*, such as *int a* or *float b*, where the programmer declares a variable. Such statements affect both the static and dynamic semantics because the compiler can perform analysis and optimisation (such as type checking) and at runtime the variable has specific attributes such as size and format. It can be thought that the programmer provides information, to the compiler, via the type. However, there is only so much that one single type can reveal, and so languages often include numerous keywords in order to allow for the programmer to express additional information.

Taking the C programming language as an example, in order to declare a variable *m* to be a character in read only memory which is accessed many times (so the compiler might consider using a register) and might be changed externally in unpredictable ways, the code *volatile register const char m* is used. Where *char* is the type and *volatile, register* and *const* are inbuilt language keywords. Whilst this keyword heavy approach works well for sequential languages, in the parallel programming domain there are potentially many more attributes which might need to be associated; such as where the data is located, how it is communicated and any restrictions placed upon it. Representing all this additional information via keywords would not only bloat the language, but could also introduce inconsistencies when multiple keywords were used together with potentially conflicting behaviours.

As first introduced in relation to the PGAS memory model in [2], the type oriented approach is for the programmer to encode all variable information via the type system by combining different types together to form the overall meaning. For instance, *volatile register const char m* is instead *var m:Char::const::register::volatile*, where *var m* declares the variable, the operator *:* specifies the type and the operator *::* combines types together. In this case, a **type chain** is formed by combining the types *Char, const, register* and *volatile*. Precedence is from right to left, so for example, the read only properties of the *const* type override the default read & write properties of *Char*. It should be noted that some type coercions, such as *Int::Char* are meaningless and so rules exist within each type to govern which combinations are allowed. It is also possible to associate arbitrary information with each type, which might be further type chains, and the type itself will give meaning to this data. To illustrate the point, if a programmer wished to suggest which register to use for variable *m* then they might express *register["ax"]* where the type itself provides the context of the string argument - in this case suggesting which register to use.

Once set in the variable declaration, the programmer can modify the semantics of a variable by changing its type later in code. Referring to the previous example, the programmer may decide to set the variable to be writable once again using the *writable* type. This can be done either permanently from that point onwards using *a:a::writable* (which appends the writable type to variable *a*'s type chain) or for a specific expression only via *(a :: writable):=99*. In such an example, if the programmer were to attempt to write to the constant variable before the writable type is applied then an error would result. In our current version of the language types must be determined at compile time but it can be very difficult or impossible for the language to support this flexible modification of types and the compiler to dynamically determine the type. For instance a conditional might rely on some user input and based upon this set the type of a variable accordingly. To ensure that types are known at compile time, typing follows lexical scoping rules and on exit from a block of code the type of a variable, if it has been modified, reverts back to what the type was when it entered that block.

The type oriented approach provides a number of advantages:

1. **Opportunities for optimisation** - Due to the programmer specifying their code in such a high level manner the compiler can obtain a much more complete view of the system and apply optimisation. The types themselves are written by domain experts which can often result in far more performant, consistent behaviour.
2. **Choice between explicit and implicit programming** - In the absence of type information the compiler will apply sensible,

Please cite this article as: N. Brown, Type oriented parallel programming for Exascale, Advances in Engineering Software (2017), http://dx.doi.org/10.1016/j.advengsoft.2017.04.006