New programming languages and paradigms

Seminar: Neueste Trends im Hochleistungsrechnen

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Autor: Lukas Stabe
Betreuer: Michael Kuhn

Hamburg, den 21.03.2016

Contents

1 Introduction

This report will cover the advantages that might be gained by introducing new programming languages and paradigms into the world of High Performance Computing.

To explain the advantages in an easily graspable way, we will follow the following structure:

First, we will define some of the terms used. Then we will enumerate the most important advantages of adopting new languages and paradigms in gerenal, and will discuss some of the problems they face. Next, we will examine some concrete examples of new languages and paradigms that are currently on the horizon of the HPC world, some already widely used, some not widely used but with a big potential for future adoption. Finally, we will end with some conclusive remarks.

2 Definitions

2.1 Programming Language

> *"A programming language is a formal constructed language designed to communicate instructions to a machine, particularly a computer."*
>
> — Wikipedia: Programming language

A programming language can be described as a definition of how the programmer instructs the machine on how to do something, for example solve a task.

How these instructions take form is completely up to the language. It can on the one extreme be in the classical imperative style where instructions written translate directly to the operations the machine uses to perform those instructions. On the other extreme, it can be an abstract way of describing the problem, with the task of figuring out a way to solve it left entirely to the machine, as is the case with e.g. logic programming.

So, in simpler words, a programming language defines how the programmer tells the computer to do something.

It should be noted that languages are often closely related to their standard library. As an example for this kind of tight coupling, types from the standard library are often granted access to special constructs that "normal" code can not use.

For this reason, the boundaries between standard library and language are often unclear, and in most cases when talking about languages, the standard libraries are implicitly included.

2.2 Paradigm

> *"A programming paradigm is a fundamental style of computer programming, serving as a way of building the structure and elements of computer programs."*
>
> — Wikipedia: Programming paradigm

A programming paradigm can be understood as a description of how a programmer should approach any specific problem.

Paradigms describe common patterns that are deemed advantageous and sometimes explicitly discourages "anti-patterns", referring to programming patterns that are seen as bad practice and leading to bad code.

To clarify this, let's explain these terms using object oriented programming (OOP) as an example of a paradigm. OOP describes the pattern of structuring an application's code in classes that are composed of their member variables and member functions. Instances of classes communicate by calling methods on each other. OOP discourages the anti-patterns of using global state and global functions instead of instance members and member functions except where it can not be avoided.

2.3 Relation between language and paradigm

> *"Capabilities and styles of various programming languages are defined by their supported programming paradigms; some programming languages are designed to follow only one paradigm, while others support multiple paradigms."*
>
> — Wikipedia: Programming paradigm

It is important to recognize that in most cases, languages and paradigms are not tightly coupled. Most languages support a mix of paradigms and in allmost all cases, any paradigm can be used in any language, although it may not be convenient and easy to do so.

Some examples of languages that support mixes of paradigms are C++, supporting object-oriented, procedural, imperative and functional programming; C is of procedural nature, but can be used in an object-oriented way with minimal hassle.

Again, standard libraries deserve a special mention, because they may be written with a specific paradigm in mind and thus might constrain where the patterns and anti-patterns of a paradigm can be respected when interfacing with the standard library.

3 Advantages and Problems

A question that might be asked when the topic of new languages in HPC comes up might be "Why?". So let us answer the question of why the high performance community should eventually move on from C or Fortran.

Of course, new languages and paradigms also come with their own problems, so we are going to have a look at those as well.

Note that these lists of advantages and disatvantages are incomplete by nature. For all the things listed here, we will provide concrete examples when talking about a few specific languages.

3.1 Advantages

The advantages new languages and paradigms bring with them mostly fall into the category of programmer-friendliness:

They can simplify development by not having difficult-to-master concepts like, for example, pointers in C. They can also simplify things like interprocess communication by providing new layers of abstraction and by automatically doing things the programmer had to manually before.

Some classes of (possibly hard to debug) errors may be completely impossble to make, since a strong type system might detect programmer errors before the code is ever executed; errors where unchecked (possibly concurrent) access to memory leads to unplanned behaviour might be impossible in a language that provides memory-safety guarantees.

Additionaly, new languages and paradigms are also likely to produce easier-to-maintain code. They may be easier to write in an idiomatic manner (especially for inexperienced programmers, which is common in HPC where scientist from other fields might learn programming to solve a specific task). This is largely a result of the communities surrounding the language or paradigms at hand. Communities of newer languages/paradigms often put a strong focus on unit-testing and documentation, and that plays a big role in producing "good" code.

Easy to maintain code is especially important in the field of HPC, since it is a common occurence that a scientist will write code to solve her problem, after which the code will then be improved and maintained by the staff operating the cluster.

One advantage that does not fall into this category of programmer-friendliness is better utilization of the resources available to the program. These resouces might be the CPU, which can be used in a more effective way due to more optimizations the compiler can make. A programm might also make use of vector units built into the CPU automatically. Another kind of resource that might be put to use are GPUs and accelerator hardware.

3.2 Problems

This section will list problems and challenges new languages and paradigms face, and suggest possible solutions to those problems.

The first challenge is that in HPC, there is a large existing codebase of C and Fortran code. This code – in part purpose-built application code and in part reusable libraries that are useful in a broad spectrum of HPC applications – is something a new language might not have. Thus, switching to a new language might mean rewriting all that code, which can be a significant amount of work.

A possible solution to this first challenge would be to provide a way for a new language to provide an interface to C and Fortran routines (a foreign function interface, or FFI) and for new paradigms to interact gracefully with code not fitting its patterns.

A second problem is a lack of tools for new languages. The HPC community has produced a large amount of specialized tools for the existing systems, and new languages might not be able to take advantage of those.

This challenge is hard to solve. In some cases, existing tools may be adapted to also work with a new language, but often entirely new tools will need to be written.

The last – and probably most important – challenge in this list is the expertise of experienced programmers in the existing languages and paradigms. People have been programming C and Fortran for over fourty years, and thus have become experts in them the likes of which can't possibly exist for a young, new language or paradigm. This challenge is especially important in HPC, since programs need to perform as best as they can, so the huge shared knowledge of how to make C/Fortran programs run fast is very valuable.

To this most important challenge, there is no easy solution. Time will produce experts in any language, but while they are young, languages and paradigms need to make up for the lost expertise by excelling in other areas.

4 Examples

4.1 Scipy

SciPy is a collection of libraries and tools for scientific computing in Python. While Python is by no means a new language – and has been popular in the scientific community for some time – its interpreted nature and resulting mediocre performance have hindered its adoption in the HPC community.

SciPy enables programmers to write faster Python applications by providing most of its functionality in so-called "extension modules". Extension modules consist of C/C++/Fortran code written to be used from within Python. All code inside the module runs at full native speed, not hindered by the Python interpreter.

So with SciPy, programmers can write their program flow and high-level structures in Python, and as long as the hotspots are contained within the SciPy libraries, the program runs at near-native speeds. The libraries are written to enable this, so they contain functionality to apply mathematical operations to large sets of numbers etc.

4.2 Rust

Rust aims to be a low-level, compiled language that is suitable for performance-sensitive tasks and as a systems language. It provides a strong type and generics system including type inference, which allows programmers to write less verbose, more concise and expressive code than in C, and at the same time have a smarter compiler that catches many kinds of errors at compile time.

The language also guarantees memory safety by disallowing memory-unsafe operations (such as dereferencing null pointers) that are not contained in a lexical scope explicitly declared "unsafe".

Through its system of read-only references, Rust also provides thread-safety as a language feature.

```
fn main() {
    // A simple integer calculator:
    // `+` or `-` means add or subtract by 1
    // `*` or `/` means multiply or divide by 2
    let program = "+ + * - /";
    let mut accumulator = 0;

    for token in program.chars() {
        match token {
            '+' => accumulator += 1,
            '-' => accumulator -= 1,
            '*' => accumulator *= 2,
            '/' => accumulator /= 2,
            _ => { /* ignore everything else */ }
        }
    }
}
```

Rust supports a variety of paradigms, functional, imperative and object-oriented among them. For HPC applications, there are multiple community-built MPI bindings currently in development.

```rust
extern crate mpi;

use mpi::traits::*;

fn main() {
    let universe = mpi::initialize().unwrap();
    let world = universe.world();
    let size = world.size();
    let rank = world.rank();

    if size != 2 {
        panic!("Size of MPI_COMM_WORLD must be 2, but is {}!", size);
    }

    match rank {
        0 => {
            let msg = vec![4.0f64, 8.0, 15.0];
            world.process_at_rank(rank + 1).send(&msg[..]);
        }
        1 => {
            let (msg, status) = world.receive_vec::<f64>();
            println!("Process {} got message {:?}.\nStatus is: {:?}",
                rank, msg, status);
        }
        _ => unreachable!()
    }
}
```

While Rust is not inherently easier to learn than C or C++, its smart compiler prevents whole classes of errors from being made, which is a big advantage. There are no conclusive performance benchmarks yet, but initial results suggest Rust might be at least as fast as C in some cases.

4.3 Swift/T

Swift/T is a language that allows programmers to write MPI applications without explicitly managing nodes, data-flow, aggregations, etc. A Swift/T script – written in a C-like syntax – does not do work by itself, but rather executes so-called "leaf-nodes" which can take many forms, including C functions, executable binaries, shell scripts, Python functions and many more. These leaf-nodes are used like functions in the Swift/T script, and the language transparently coordinates the data flow between those leaf-nodes.

```
int X = 100, Y = 100;
int A[][];
int B[];
foreach x in [0:X-1] {
  foreach y in [0:Y-1] {
    if (check(x, y)) {
      A[x][y] = g(f(x), f(y));
    } else {
      A[x][y] = 0;
    }
  }
  B[x] = sum(A[x]);
}
```

Due to the way Swift/T works – by building a graph of the input and output data of all the leaf-nodes – it can easily run single nodes concurrently when possible. In the examples above, all iterations of both loops run concurrently, and in addition, both the calls to `f()` run concurrently as well.

4.4 OpenMP 4

OpenMP is not a new citizen in the HPC ecosystem. It is a set of compiler directives working on top of C, C++ and Fortran that helps with parallelizing programs easily. It is a popular tool to implement multicore-parallelism.

The new version, called OpenMP 4, has added (among other changes) a few interesting new directives that allow better utilization of hardware.

The new SIMD directive allows programmers to transparently use vector units like AVC/SSE and NEON to do numeric operations on multiple sets of numbers in parallel on one core. It also works combined with the PARALLEL directive to split the data to process by core first, and then utilize vector units to process multiple data points on each core.

The TARGET directive simplifies the use of accelerators. Accelerators are hardware that is built for the specific purpose of processing large amounts of numeric data in parallel. Without OpenMP 4, using these meant either using proprietary software made by the hardware vendor or rely on difficult to use software like OpenCL. OpenMP 4 TARGET directives automatically run the code contained within them on the accelerator hardware and automatically transfer input and output data to and from the device.

```
void vadd_openmp(float *a, float *b, float *c, int len)
{
    #pragma omp target map(to:a[0:len],b[0:len],len) map(from:c[0:len])
    {
        int i;
        #pragma omp parallel for
        for (i = 0; i < len; i++)
            c[i] = a[i] + b[i];

    }
}
```

5 Conclusion

In conclusion, new languages and paradigms can provide big benefits in almost any aspect important to HPC. They can simplify development, produce easier-to-maintain code and help programmers to easily utilize new and existing resources.

On the other hand, these new languages need to overcome some significant challenges. A large existing codebase and ecosystem need to be replaced or adapted and the established languages lead in performance, which is a gap that needs to be closed.

So while there seems to be nothing that can seem to replace C, C++ or Fortran right now, it is only a matter of time until a new language gets its turn to be "the" language in HPC.