Introducing Tetra: An Educational Parallel Programming System

Ian Finlayson, Jerome Mueller, Shehan Rajapakse, Daniel Easterling
The University of Mary Washington
Fredericksburg, Virginia, USA
finlayson@umw.edu, {jmueller, srajapak, deast3cx}@mail.umw.edu

***Abstract*—Despite the fact that we are firmly in the multicore era, the use of parallel programming is not as widespread as it could be – in the software industry or in education. There have been many calls to incorporate more parallel programming content into undergraduate computer science education. One obstacle to doing this is that the programming languages most commonly used for parallel programming are detailed, low-level languages such as C, C++, Fortran (with OpenMP or MPI), OpenCL and CUDA. These languages allow programmers to write very efficient code, but that is not so important for those whose goal is to learn the concepts of parallel computing. This paper introduces a parallel programming language called Tetra which provides parallel programming features as first class language features, and also provides garbage collection and is designed to be as simple as possible. Tetra also includes an integrated development environment which is specifically geared for debugging parallel programs and visualizing program execution across multiple threads.**

***Keywords*-Parallel Programming; Education; Debugging;**

I. Introduction

We are firmly in the multicore era. Nearly all general-purpose computing devices (including mobile phones) have more than one processor core. Furthermore, processor clock speeds have largely levelled off. Unable to significantly increase single-core performance, processor designers place more cores in each chip. In order to achieve increased performance, we will have to employ parallel programming. Unfortunately this has yet to happen in a truly widespread way.

One reason for this is that parallel programming tends not to be a major part of undergraduate computer science education, despite repeated calls for it to be introduced earlier and more thoroughly [1] [2] [3].

One obstacle present in teaching parallel programming to undergraduates is the programming languages commonly used for parallel programming. Because the main reason to employ parallel programming is increased performance, the languages commonly used for parallel computing are those focussed on performance: C, C++, Fortran, OpenCL and CUDA. Unfortunately, these are difficult programming languages even for many upper-level students to use effectively. Coupled with the shift towards simpler languages for introductory computing classes, it is difficult to use these languages to introduce parallelism early in the computer science curriculum.

Commonly used introductory languages like Python and Java can also be used for parallelism of course, but this is also not quite ideal since these languages do not include parallel concepts as native, first class language features as OpenCL and CUDA do. Nor do they have systems like OpenMP which adds parallelism on top of C, C++ and Fortran. Of course these languages have parallelism in the standard library, but there are advantages to including parallelism as a first class language feature, as we will discuss in Section 2.

In particular, Python has recently become the most popular introductory programming language at top U.S. schools [4]. While generally a fine language, Python is especially poorly suited for parallel programming due to the fact that the canonical implementation uses a global interpreter lock (GIL) [5]. This means that, in a multi-threaded Python program, only one thread can actually run at a time. It is possible to write *concurrent* programs in Python, but one cannot achieve speedup with a truly *parallel* program [1].

Another obstacle to learning parallel programming is the lack of debuggers or integrated development environments (IDEs) specifically geared to parallel computing. Those that exist are typically difficult to use, proprietary, or lack features that would help programmers debug parallel code. This is especially a problem as debugging parallel code is much more difficult than sequential code due to the possibility of race conditions and deadlock.

This paper introduces a new programming called Tetra which attempts to address these issues by providing parallel programming features as first class language features. The syntax of the language is similar to Python, and the language is designed to make programming both sequential and parallel programs as simple as possible. As an educational language, Tetra places a higher emphasis on simplicity than performance or generality. Tetra also includes an integrated development environment which is specifically geared for debugging parallel programs and visualizing program execution across multiple threads.

The rest of this paper is organized as follows: Section two discusses the design of the Tetra programming language,

[1] It is possible to write parallel Python programs by running more than one interpreter, and it is also possible for code in Python modules (written in C) to exploit parallelism, but neither of those approaches is a natural way to teach parallel computing.

including both its conventional language features and those that support parallelism. Section three discusses the design of the Tetra IDE. Section four briefly discusses the current implementation of the language and IDE. Section five reviews related work in parallel programming languages and debuggers. Section six discusses our plans for future work and section seven draws conclusions.

II. Language Design

In this section we will discuss the design of the Tetra programming language. Besides the support for parallelism, Tetra is a relatively simple procedural programming language. It borrows much of its syntax from Python, including many keywords, comments beginning with a pound sign, and the use of white-space and colons to delimit code blocks.

One difference from Python is that Tetra is statically typed: all types are known at compile/parse time. Like several other newer languages, Tetra does have type inference for local variables. This means that most variables do not need type declarations; the interpreter infers the types from the usage of the variable. Function parameters and return values do need to have declared types, however.

Tetra's primitive types currently include integers, reals, strings and booleans. Tetra also includes arrays of these types (including multi-dimensional arrays). Tetra's control structures include if statements, while and for loops, and function definitions. Figure II lists a simple Tetra program which calculates a recursive factorial.

```
# a simple factorial function
def fact(x int) int:
  if x == 0:
    return 1
  else:
    return x * fact(x - 1)

# a main function which handles I/O
def main( ):
  print("Enter n: ")
  n = read_int( )
  print(n, "! = ", fact(n))
```

Figure I. A Simple Sequential Program

As discussed in the future work section, there are several additional language features we would like to include, but the core language was kept very simple initially to decrease the amount of development time to a working product.

As mentioned, Tetra has parallel programming constructs built-in as first class language features. The simplest of these is the `parallel` statement which has the following form:

```
parallel:
  <statement 1>
  <statement 2>
  ...
  <statement n>
```

Each of the statements under the parallel block will be run in parallel, separate from the others. The program will then wait for all n statements to finish before moving on to any code after the parallel block. Each of these statements can be any legal Tetra statement including a function call, assignment or loop.

This makes it easy to write multi-threaded programs. The programmer need only specify that a set of statements should be done in parallel, and the language takes care of creating, starting, and joining the threads.

Figure II provides a listing of a Tetra program uses a parallel statement to calculate the sum of the first 100 natural numbers in two threads.

```
# sum a range of numbers
def sumr(nums [int], a int, b int) int:
  total = 0
  i = a
  while i <= b:
    total += nums[i]
    i += 1
  return total

# sum an array of numbers in parallel
def sum(nums [int]) int:
  mid = len(nums) / 2
  parallel:
    a = sumr(nums, 0, mid - 1)
    b = sumr(nums, mid, len(nums) - 1)
  return a + b

# print the sum of 1 through 100
def main( ):
  print(sum([1 ... 100]))
```

Figure II. A Parallel Sum Program

Tetra also provides `background` blocks which have the following form:

```
background:
  <statement 1>
  <statement 2>
  ...
  <statement n>
```

Background blocks are similar to parallel blocks, except that they do not wait for all of their sub-statements to terminate before moving on to the rest of the program. They can be used to launch tasks in the background.

Tetra also provides a parallel for loop which has the following form:

```
parallel for <var> in <sequence>:
  <statement 1>
  <statement 2>
  ...
  <statement n>
```

Syntactically, it is identical to the regular for loop except for being prefaced with the `parallel` keyword. Semantically, the difference is that each loop iteration can be done independently of the others. Thus Tetra makes it very easy to parallelize an algorithm which consists of loops with no inter-dependence.

Of course not all algorithms are trivially parallelizeable. Sometimes it is necessary to utilize mutual exclusion to prevent multiple threads of execution from interfering with each other. Tetra allows this in the form of `lock` statements which have the following form:

```
lock <name>:
  <statement 1>
  <statement 2>
  ...
  <statement n>
```

A lock works by associating a lock name with the piece of code in the block. Lock names are in a separate namespace from other Tetra identifiers, so the name can be an existing variable name, or any other legal identifier name. When a Tetra program gets to a lock block, it checks that no other thread of execution is in a lock block that has the same lock name. This can be the same block, or a different lock block with the same name. It waits until all threads have exited the lock block, then proceeds to execute it which keeps other threads from doing so too.

Figure II lists a Tetra program which uses a parallel for loop and a lock to find the largest value in an array of numbers.

Because multiple threads could update the `largest` variable at the same time, it is protected inside of a lock. We also must check again if the number is still the largest inside the lock since it is just possible that the largest value was replaced between the previous check and entering the lock.

The parallel features of Tetra are simple and somewhat limited for real-world use, but they allow for expressing many parallel algorithms, and introducing ideas such as race conditions, deadlock, and other issues beginners to parallel computing must contend with.

Having these parallel programming constructs as first-class language features has several advantages. First off, it minimizes the overhead of parallelization. Code oftentimes would not need to be restructured to be parallelized; it can just be wrapped in the appropriate blocks.

```
# find the max of an array
def max(nums [int]) int:
  largest = 0
  parallel for num in nums:
    if num > largest:
      lock largest:
        if num > largest:
          largest = num
  return largest

# run it on some numbers
def main( ):
  nums = [18, 32, 96, 48, 60]
  print(max(nums))
```

Figure III. A Parallel Max Program

Secondly it avoids programming errors caused by forgetting to do something necessary to support threading in a typical language. For example, if one forgets to join a thread, a race condition can be introduced. Likewise, if one forgets to unlock a mutex, a deadlock situation can be introduced. Tetra makes such mistakes less likely because the programmer does not need to do those things manually.

It also signifies that parallel computing is not an optional "add-on", but a core part of the programming language. In the future, parallel programming may not be the relatively specialized sub-field of programming that it is now, and our programming tools should reflect this.

III. Integrated Development Environment

While having parallel language constructs built in to Tetra should make it easier for beginners to learn parallel programming, having a suitable environment may make even more difference. To that end, Tetra includes an Integrated Development Environment (IDE) geared towards developing and debugging parallel programs.

Figure III shows a screen shot of the current state of the IDE. Currently supported features include basic editing features (loading, saving, copy and paste etc.) as well syntax highlighting of Tetra keywords and the ability to run Tetra programs from the IDE. Like many other IDEs, program input and output are directed to a console pane.

We are currently working on adding the ability to step through Tetra code in the IDE as the program is running. Unlike most debuggers, the Tetra IDE will have multiple code views in debug mode: one for each thread of the currently running program. This will allow students to step through the different threads independently.

This ability will help students discover race conditions and deadlock scenarios by stepping through the code in different orders. For example, they can step though the code in one thread all the way to the end (or a lock) to ensure that this does not negatively impact what the other threads are doing.

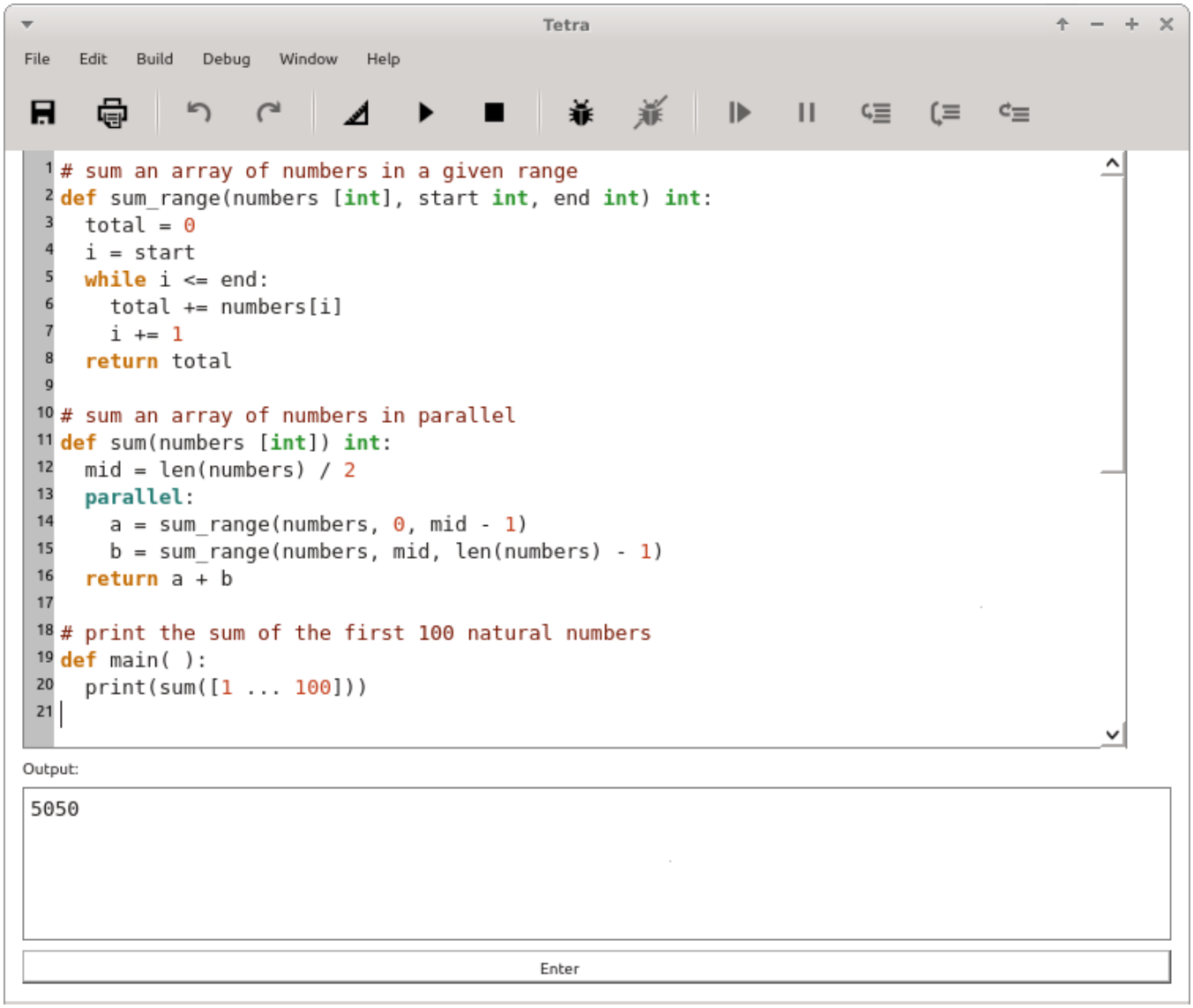

Figure IV. The Tetra IDE

While obviously a work in progress, we believe these features of the IDE will be extremely valuable in learning and teaching parallel programming.

IV. Implementation

The Tetra system currently consists of three major components: the compiler front-end, the interpreter, and the IDE. This section discusses the implementation of this system.

The compiler front-end is written in C++ with the help of the Bison parser generator [6]. The parser reads in Tetra source code, and parses it into a abstract syntax tree (AST). The lexical analyzer is hand-written which was necessary to handle the significant white space in Tetra.

After the code is parsed into an AST, it has type checking and type inference applied to it. Because type inference is only done on the local scope, a simple flow-based algorithm suffices. If any type errors are found, then they are signalled.

The interpreter is a C++ library which invokes the front-end to parse the code it is given into the AST. Once this is done, it interprets the code by traversing the AST recursively. For the most part, this is identical to the way other tree-based interpreters work.

However, when the Tetra interpreter gets to a node in the AST which represents a parallel block, it launches one thread for each child node (using the Pthreads library) and executes them in parallel. It then waits for each of those threads to join before moving on. This is also done for background blocks except it does not join the threads which were spawned.

Parallel for loops are handled in a similar way, except that each thread needs to have its copy of the induction variable

inserted into its private symbol table. Because of the way threads are created dynamically, they have private and shared symbol tables.

Lock statements are handled with Pthread mutexes. When a thread enters a lock node, it locks the corresponding mutex. When it exits, it unlocks the mutex. This way, only one thread of execution can execute a critical section at a time.

The interpreter is written as a library in order to facilitate exposing its features to the IDE. There is also a command line driver program for it which simply calls the interpreter on its argument from start to finish.

The IDE is also written in C++ using the Qt library for the user interface. It calls upon the interpreter library to allow the user to run programs. We are currently working on exposing more of the interpreter's internals in the library so that the IDE can do program inspection and control execution – by stepping through the code line by line.

A lot of effort was put into ensuring that the interpreter actually provides speedup when given a parallel program. Due to the sharing of data structures amongst interpreter threads, this was not easy to achieve – which could explain why interpreters for Python and other popular languages avoid this issue altogether. To test the speedup we used two Tetra programs: one which calculates the first million primes, and one which solves an instance of the travelling salesman problem. Each of these programs achieves approximately 5X speedup when run on 8 cores which is a 62.5% efficiency rate. Thus Tetra programs are able to achieve speedup when running parallel programs, but more can be done to improve the efficiency of the interpreter.

V. Related Work

This section discusses other works similar to Tetra, including both programming languages geared towards parallelism and parallel programming debuggers.

There are several programming languages focussed on parallel programming, and making parallelism easier to express. Among these are X10 [7], Chapel [8], ParaSail [9], Cilk [10] and Unified Parallel C (UPC) [11]. Tetra is similar to these languages in that they all include parallel programming features at the programming language level. Tetra is different in that it is aimed squarely at beginning programmers as opposed to those in the parallel computing or high performance computing fields.

There has also been prior work in the area of debugging parallel applications. Many debuggers such as DDD [12], Eclipse [13] and Visual Studio [14] have some ability to debug multi-threaded applications. Typically one can inspect the program state of each thread, but features planned for Tetra such as the ability to step through threads independently is not available. This is because these debuggers work by inspecting the state of a native application, which does not provide this facility.

There are two proprietary debugging solutions specifically geared towards easily debugging parallel programs: TotalView from Rogue Wave Software [15], and Allinea DDT [16]. Both of these are expensive proprietary products, and are geared towards professional programmers in the field, not at beginning students.

VI. Future Work

In this section, we discuss some of the work we have planned for the future of Tetra. The most pressing piece of future work to accomplish is the completion of parallel debugging features in the IDE. Besides that, there are several other enhancements planned.

The language was intentionally kept very simple to reduce the amount of initial work needed to reach a working product. There are several other desirable language features such as the ability to create new types with a class statement, the inclusion of additional built-in types such as associative arrays and tuples, and error handling that we would like to add to the language to increase its expressivity.

The language also has an extremely spartan standard library at the moment, which consists only of basic I/O functions and functions for finding the lengths of strings and arrays. A more robust library with mathematical functions, string handling functions and so on will be developed in the future as well.

Another important piece of future work is teaching a parallel programming class in which Tetra is used. This will thoroughly test the language, IDE, and interpreter, as well as allow us to gauge how effective Tetra is for instructing students in parallel programming techniques which will guide its evolution.

We also plan on evaluating the effectiveness of Tetra empirically through a pedagogical study in which it is used for instructing beginners in parallel computing, as compared to other languages.

Lastly, we plan to add a native code compiler, which will compile Tetra code into an efficient executable, possibly by targeting C with Pthreads as the output language. This will allow Tetra programs to be run more efficiently than with the interpreter. With this in place, one could write a Tetra program, run it through the IDE and step through it in the debugger when it is being developed, then compile it to a native executable to run it more efficiently. It would even be possible to target CUDA or OpenCL so that compiled Tetra code could run directly on a GPU.

VII. Conclusion

There is currently a disparity between the programs we write and the machines we run them on. Computer systems are increasingly parallel machines while, by and large, the programs we run on them are sequential. If we are going to see increased program performance in the future, we

will need a shift in computer science and computer science education towards embracing parallel programming.

Another trend in programming and computer science education, is one towards higher-level languages such as Python. These languages are easier for beginners to learn, but are not commonly used for parallel computing. Historically this is because people who write parallel programs are those who are interested in maximizing performance and use C, C++ or Fortran for that reason.

In a many-core world, however, exploiting parallelism effectively will be more important in terms of performance than the gains associated with using an efficient low-level language.

Tetra attempts to address this disparity by including parallel programming facilities as first class language features, and by providing a debugger to make the development and testing of parallel programs as easy as possible.

References

[1] Y. Ko, B. Burgstaller, and B. Scholz, "Parallel from the beginning: The case for multicore programming in thecomputer science undergraduate curriculum," in *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '13. New York, NY, USA: ACM, 2013, pp. 415–420. [Online]. Available: http://doi.acm.org/10.1145/2445196.2445320

[2] L. Ivanov, H. Hadimioglu, and M. Hoffman, "A new look at parallel computing in the computer science curriculum," *J. Comput. Sci. Coll.*, vol. 23, no. 5, pp. 176–179, May 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1352627.1352655

[3] L. R. Scott, T. Clark, and B. Bagheri, "Education and research challenges in parallel computing," in *Proceedings of the 5th International Conference on Computational Science - Volume Part II*, ser. ICCS'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 44–51. [Online]. Available: http://dx.doi.org/10.1007/11428848_6

[4] P. Guo. (2014) Python is now the most popular introductory teaching language at top u.s. universities. [Online]. Available: http://cacm.acm.org/blogs/blog-cacm/176450

[5] D. Beazley, "Understanding the python gil," in *PyCON Python Conference. Atlanta, Georgia*, 2010.

[6] C. Donnelly and R. Stallman, "Bison. the yacc-compatible parser generator," 2004.

[7] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An object-oriented approach to non-uniform cluster computing," *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094852

[8] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[9] S. T. Taft, "Multicore programming in parasail," in *Reliable Software Technologies-Ada-Europe 2011*. Springer, 2011, pp. 196–200.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, *Cilk: An efficient multithreaded runtime system*. ACM, 1995, vol. 30, no. 8.

[11] T. El-Ghazawi and L. Smith, "Upc: unified parallel c," in *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM, 2006, p. 27.

[12] A. Zeller and D. Lütkehaus, "Ddda free graphical front-end for unix debuggers," *ACM Sigplan Notices*, vol. 31, no. 1, pp. 22–27, 1996.

[13] E. Foundation. (2014) Eclipse ide. [Online]. Available: https://eclipse.org/ide/

[14] M. Corporation. (2014) Visual studio. [Online]. Available: http://www.visualstudio.com/

[15] I. D. I. Solutions, "Totalview debugger: A comprehensive debugging solution for demanding multi-core applications," Retrieved 03/04/2009, from http://www. totalviewtech. com/pdf/TotalView Debug. pdf, Tech. Rep., 2009.

[16] K. Antypas, "Allinea ddt as a parallel debugging alternative to totalview," *Lawrence Berkeley National Laboratory*, 2007.