Intro

Hello everyone, I want to thank you all for being here. My name is Caleb Huck and the title of this presentation is OpenMP for Python.

Organization

Add later

Background

So, let’s start with some background. In terms of hardware, almost everything today is parallel. From budget laptops all the way up to supercomputers and even much of embedded computing, microcontrollers, everything is parallel. And we’re only moving more and more in the direction of parallelism, which is logical, because 1) we get more computing power, and 2) the real world is parallel. Things happen at the same time, independently of each other, that’s normal. So it follows that the computing paradigm would naturally move in this direction.

Background 2

So, what are the practical implications of the shift towards parallelism in computing? Well, in the industry, there is a huge demand for computer scientists and programmers who have the skills to work with parallel systems whether directly or indirectly. Not every programmer will be writing parallel code or anything performance oriented necessarily. But they still need to have understanding of the systems that their code will run on and be able to think in terms of parallelism.

The problem is that, in academic environments, the curricula is still emphasizing serial coding and thinking which obviously needs to change. And it is, the newest ACM and ABET standards are calling for parallel concepts to be taught earlier in the coursework, even in introductory CS classes. And they’re being taught in required classes rather than just in electives as they were in the past. And there has been a lot of recent research exploring the best way to start integrating this new material into the existing curricula, I’ve included just a few of them in the references but there are many more.

IPDC

iPDC is an initiative here at Tennessee Tech that focuses on providing professors with resources and training on how to teach parallel concepts in early CS courses. Professors can attend one of the workshops where they go through some basic lectures on parallel concepts and fundamentals and then participate in hands on programming exercises. There are also teaching modules on the iPDC website that anyone can access and use and include programming exercises as well as what we call “unplugged” activities where parallel concepts are taught without using programming or computers. And these are great for getting students to connect the concepts to real life and think in parallel because that really is our main goal, to teach them how to think in terms of parallelism and concurrent dependencies not just get really good at writing threaded code or using some multithreading library (sort candy example?)

IMG slide

I ran across this image while watching a conference presentation talking about Python asyncio and ironically enough why the Python GIL is a feature instead of a limitation. If you don’t know what the GIL is ill cover that more later. But the sentiment of this photo isn’t completely without merit. Writing multithreaded code is hard, for several different reasons. One of them is that thinking in parallel doesn’t always come naturally particularly to new programmers and everything you have to keep up with in your mind can quickly get complicated and difficult to manage. So, it is important to keep this in mind when choosing a tool for teaching parallelism.

OpenMP

OpenMP or Open Multiprocessing is a shared-memory multithreading library that we use in every IPDC workshop. It was first released for FORTRAN in 1997 as a 60 or so page document and has since gone through many revisions. The latest version is up to over 700 pages and has added lots of new features and abilities. The most important thing that OpenMP does is that it removes the responsibility from the programmer for thread creation and management. This is great because the programmer can focus on how they want their code to be run in parallel without all the extra overhead of writing the low-level threading code. Instead, they describe how the code should be run using high level directives and clauses and the code is transformed by the preprocessor into the equivalent threaded code before the compiler ever sees it. That image at the bottom there shows just how little boilerplate code is required to have a basic parallel region. That simple directive will automatically create threads and execute the next block in parallel and that’s it. This is very useful for teaching parallel concepts to new students because they are better able to focus on the concepts themselves instead of putting their effort towards learning the ins and outs of a threading library.

Problem

So, what is the problem? Right now, the top three languages used for teaching early CS classes is C or C++, Java, and Python. And they all have roughly equal market share across universities. Out of these, OpenMP only supports C/C++. There have been a few projects that implemented OpenMP in Java with varying levels of success. But to our knowledge, there haven’t been any implementations in Python.

Contribution

So, that is exactly what we have been working on. The goal of this project has been to implement core OpenMP directives, clauses, and runtime functions, for python. And I’ll explain what I mean by core on the next slide. But I want to emphasize that this is a teaching tool, not a real-world performance library. So we want to be able to demonstrate the effects of applying parallelism to different problems, but we certainly don’t expected to achieve the same kind of scaling or efficiency that we could get with C. Our goal is to have the same benefits of OpenMP for teaching concepts like speedup, efficiency, and synchronization, but in the Python language.

OpenMP Core

As I mentioned earlier, the current OpenMP specification has grown tremendously, however, the core of OpenMP has pretty much been around since the beginning. These are the directives, clauses, and runtime functions that make up most of OpenMP programs and are sufficient to write almost any parallel algorithm. I should mention that the one thing we are missing with this core is the ability for task parallelism, which would be done with the sections and section directives. This isn’t critical for our teaching purposes, which is why they aren’t included now but we do plan to add them later.

Related Work

There have been several projects in the past that implemented OpenMP in Java. Each of these had problems with usability and sometimes correct function, and we learned a lot from using them and exploring their source code. The problems we had with them helped form a basis for our usability goals in order to have a software product that is easy to use for early CS students and professors so that it can be a practical teaching tool. So that is why I have included them here and I’ll just cover them briefly. Jomp was the first project in 2000. It required the user to write their code in a .jomp file, which was standard java code with the OpenMP directives as single line comments. Then they would run it through the jomp preprocessor to generate .class files which could be compiled and run normally. Pyjama made the process simpler by combining the preprocessor with the compiler, so you could run one tool to generate the class files. Pyjama seems to be the most complete and usable of the three which is why it is used in the java portion of the iPDC workshop. Finally Omp4j was the latest project and was quite similar to pyjama except for the fact that they didn’t implement a runtime library. Instead they had some preprocessor-level macros that seemed to break in many circumstances and didn’t work very well.

Design

So, like I said we learned quite a bit from the Java versions and even though Python is a different language, we draw a lot of our design goals from our experience with them. First, we want our software to be easy to download, install, and get started using. This is important since for one, we expect many of our users to be early CS students and if the software is hard to use or complicated to install correctly, then they will have a lot of problems and then professors have to spend their time helping them get it corrected. Also, professors tend to be busy people and many of them may not have the time to troubleshoot and figure out how to use the software if it isn’t straightforward. We also want to provide helpful error messages. One of our main frustrations with the Java versions was that errors that got through the preprocessor would show up as java compiler errors from the transformed code and had nothing to do with the user code. Also, since one java file would generate multiple class files, the error you get isn’t even from the same file which makes debugging difficult. Another of our goals, is to be able to use our tool effectively and easily with IDEs. Lastly, we want to maintain as much familiarity with original OpenMP as possible in terms of syntax and style.

Python Obstacles 1

As you might imagine, we faced some problems making OpenMP work in Python. Python is very different from C in terms of syntax and programming paradigm. First, it’s interpreted instead of compiled. Also, whitespace and indentation matters in Python and there are no primitive types, everything is an object. So here are three of biggest problems we had to deal with early on in the project.

Python Obstacles 2

The first was the global interpreter lock. In CPython, which is the most common interpreter, anytime a thread accesses any shared resource, the interpreter is locked for all other threads. This helps prevent things like race conditions but it also effectively serializes your multithreaded code. To solve this problem, we use the jython interpreter. Jython is a java-based python interpreter that was implemented without a global interpreter lock, and uses java threads under the hood.

Another problem we faced has to do with data structures. In C, we use arrays a lot with OpenMP. But in python, there is no primitive array type, instead we have list objects. And the problem with lists is that even in jython, they are automatically locked for both reading and writing, even if you’re reading from or writing to different index locations in the list. We found a partial solution to this problem. Jython includes a module called jarray that has two functions, array and zeros. These basically return a java array wrapped in an object with almost identical member functions and usage to lists. Using these we can at least get parallel reading from the array, but writing is still serialized unfortunately. Those two graphics show the difference in how you create a jarray list to a regular list. After you create them, they work pretty much exactly the same

Lastly, in python, there is no way to define an arbitrary new scope. In C you can use curly braces to create a new scope wherever you want, but in python there is no syntax to do that. To fix this, we changed the python syntax to allow an OpenMP comment to come before an indented block. The only downside to this is that we lose compatibility with any standard python interpreter if the code isn’t run through our preprocessor first, because it will appear as a comment followed by an indented block, which is obviously illegal. We felt it was worth it to make this compromise to keep consistent with python indentation with our OpenMP directives.

Architecture

Here is a quick overview of how the original code is transformed into threaded code. We use the ANTLRv4 parser generator to lex and parse the user’s code. Then our custom visitor traverses the AST and prints the transformed code to the SourcePrinter object. Because of some nice features in the way Python works, we are able to get all the information we need and transform the code in a single pass over the syntax tree. The output is the multithreaded code that depends on our backend runtime library. The backend runtime handles things like running parallel targets with the correct number of threads and managing OpenMP for loops and a few other things related to how the OpenMP blocks are executed. And then of course we have our user runtime that only has two functions for now, omp\_get\_thread\_num() and omp\_get\_num\_threads() which were on that slide from earlier.

Parallel directive

Here is a quick syntax example of a basic parallel directive and the C++ equivalent. You can see that there is really just a lot less boilerplate code in the python version, but other than that they are almost identical.

For Directive

I also wanted to quickly cover the for directive. Python doesn’t have for loops like in C. Instead, it only has for-each loops. So to deal with this, we put restrictions on the syntax of a for loop following a for directive. So, if you look at the example here, it starts with for, then a single variable, in, and then the next thing has to be range, and range can take 1, 2, or 3 parameters which basically correspond to the three parts of a C for loop. Here the range will be from 0 to 99, in increments of 2, so the output will be exactly the same as the C++ loop below it. If there are only two parameters, then the third is assumed to be 1, and if there is only one parameter, then the range is assumed to start at 0 and go until whatever parameter is passed.

Performance

Okay so we tested our python OpenMP using a very simple benchmark, it’s just simple matrix multiplication between two square matrices. You all probably know this an embarrassingly parallel problem, which means that the work of each thread doesn’t depend on results from any other thread, their work is independent of each other. Also, no threads needs to write to the same memory location, although that’s only kinda true because of the way python data structures work as I talked about already. So, we ran the benchmark with 1, 2, 4, 6, 8, 10, and 12 threads with n equal to 100, 200, 300, 400, and 500.

Runtime

Here are the runtime results. As you can see, we start to get diminishing returns pretty fast with higher thread counts. This isn’t a surprise because the more threads that are sharing the work, the more contention they have when writing to the resulting matrix jarray, since writing is synchronized even if it’s to different locations in the.

Speedup

The speedup chart is a little more interesting. If you notice the bottom blue line that represents two threads, we actually get superlinear speedup for smaller problem sizes. Also, for higher thread counts, speedup tended to go down from 400 to 500 for some reason. The only explanation I have for this right now is that unexpected things happen when you have this many layers of abstraction on the hardware. By that, I mean that we’re running a python interpreter on top of the JVM which runs on the hardware. So there is a lot in the middle where things are happening and its hard to know exactly what.

Efficiency

Efficiency is somewhat the same story. It kinda trended up like we would expect with larger problem sizes but not really. And for two threads it dropped from 1.2 down to 1 from the superlinear speedup.

Conclusion

In conclusion, we have implemented core OpenMP directives, clauses, and runtime functions in python. We fit the OpenMP style and syntax to the python way of doing things and tried to strike a good balance between the two with some strategic compromises. We tested our python OpenMP using n x n matrix multiplication and the results were more than adequate for our purpose of teaching parallel concepts to early CS students.

Questions

Thank you, are there any questions?