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

Here is the organization for the rest of the presentation. I’ll start with background on the project and the motivation for it. Then I’ll cover the contribution and related work. Then I’ll talk about the design and software architecture and the specifics of how it works. After that I’ll show a quick demo of how to run a couple small program. And last will be the microbenchmarks we used to test performance and conclusions and future work.

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, it’s all 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, and that’s normal. So, it makes sense 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 is going to 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 research exploring the best way to start integrating this new material into the existing curricula, I’ve included a few recent examples in the references but there a lot 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.

IMG slide

I ran across this image while watching a conference presentation talking about Python asyncio and ironically enough why the Python global interpreter lock is a feature instead of a limitation. If you don’t know what the global interpreter lock is, I’ll 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 head 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 is used in the IPDC workshops. It was first released for FORTRAN in 1997 as a 60 or so page document and has since gone through quite a few 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 compiler directives and the code is transformed into the equivalent threaded code automatically. 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? Currently, 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, and I’ll talk more about those in a minute. But to our knowledge, there haven’t been any implementations in Python un until now.

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, in 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. And Python isn’t a performance language. We want to be able to demonstrate the effects of applying parallelism to different problems, but we certainly don’t expect 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. And when I say core, I mean the parts of OpenMP that make up most of OpenMP programs and are sufficient to write almost any parallel algorithm. This slide shows the core that we have implemented. 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. We didn’t consider them critical for teaching purposes so that’s why it isn’t included yet, but we do plan to add that at some point 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 the java files which could be compiled and run normally. So there were a lot of steps involved. Pyjama made the process simpler by combining the preprocessor with the compiler, so you could write your code in .java files and then 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 workshops. 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 really 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’s code which makes debugging really difficult. Also, since one java file would generate multiple class files, the error you get isn’t even from the same file which makes it even more 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. It’s interpreted instead of compiled. Also, whitespace and indentation matter in Python and there are no primitive types, everything is an object. So, there were three primary obstacles I want to talk about that we had to deal with early on in the project.

Python Obstacles 2

The first was the global interpreter lock that I referenced earlier. In CPython, which is the most common interpreter used today, 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 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.

Python Obstacles 3

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 return a Python object that wraps an underlying Java array and has 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. The two code examples on this slide do the exact same thing and show the difference in how you create a jarray list compared to a regular list. After you create them, they work pretty much exactly the same.

Python Obstacles 4

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 in our parser to allow an OpenMP comment to come before an indented block, which is how you define a scope in Python. The only downside to this is that we lose compatibility with standard python interpreters if the code isn’t run through our preprocessor first, because, as you can see, it will appear as a comment followed by an indented block, which is illegal in Python. We felt it was worth it to make this compromise because it lets us stay consistent with Python indentation with our OpenMP blocks and should be more intuitive to users.

Software Architecture

Here is a high-level overview of the architecture of the software. The bright green-blue arrows show the path of the source code through the program. As you can see, it involves two operating system-level processes running at the same time. One is a java process, and the other is a Python 3 process. Early on in the project, we started working on the preprocessor and experimenting with code transformation before we settled on a solid direction for the project. One of the limitations of Jython that made us hesitate to use it is that it only supports Python 2.7, so we explored a lot of other options to see if we could find a way to use Python 3, but unfortunately none of them panned out. So, by the time that we settled on using Jython, we had already committed to writing the preprocessor in Python. So that’s why there are two processes. We used the Java ProcessBuilder API to start the Python process from Jython and pass the file name as an argument. Inside the preprocessor, we use ANTLR 4 to lex and parse the source code. After that, we use the visitor design pattern to traverse the abstract syntax tree and generate the multithreaded code. If all of that completes without an error, the multithreaded code is sent to stdout. Then on the java end, stdout and stderr are captured and if there was an error, then the error message is displayed and the program exits, otherwise, the transformed code is written to a temporary file and then executed. While it’s running, the code relies 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. I’ll show some examples of how that fits in in the next couple of slides. So, finally, there’s the user runtime and that contains the OpenMP core functions from the earlier slide.

Parallel Directive

Before I go into how the code is translated from OpenMP to threaded code, I want to briefly show the difference side by side how it looks in python compared to the original in C++. These code blocks do the exact same thing. They both just print a simple message saying hello from the thread’s id. Syntactically, the only thing we changed in the Python version is that you don’t have to put the “pragma” at the beginning of the directive. In C++ pragma comes before a compiler directive, but since it has no meaning in Python, we just start with omp.

For Directive

Implementing the for directive in python is a little less straightforward. Python doesn’t have the same for loop structure that C++ uses. Instead, python has for each loops. So, to make it work, we enforce restrictions on how the for loop following a for directive has to be structured. It has to look like this, for, and then a single variable. And then after the in, the next thing has to be the range function. And range can take 1, 2, or 3 parameters. If there are 3 given, the <NEXT> first is where the range will start, the <NEXT> second is the value to count up to, and the <NEXT> third is the step, so that’s the value that we increment by going through the range. You can see that functionally these values correspond to the three parts of a C++ for declaration. So again, <NEXT> we start at zero, <NEXT> we go as long as i is less than 100, and each iteration, <NEXT> we increment i by two. So, this may be a little weird at first if you don’t have experience with python, but in general, this is how you replicate the behavior of a for loop in python.

Source-to-source Translation

Here is an example of how an OpenMP parallel directive is translated by the preprocessor. The purpose of the parallel directive is to spawn new threads and execute the following block in parallel. First, if the num\_threads clause is not present, then the number of threads that get spawned will default to the output of os.cpucount(), which is just a python standard library call. In this case it was 8 because my laptop has 8 virtual cores. Also, the reason that the variable starts and ends with an underscore is because it was created by the preprocessor. This is to prevent collisions with user variable names. So, we make it a requirement that user variables can’t start and end with an underscore. This is the target function declaration that gets passed to the threads to execute. This line shows how the shared clause works. The global keyword binds the var1 from the outer scope to the inside scope, making a shared variable. This line is the private clause. Python has what is called the copy on write rule. The way it works is that any variable from the outer scope is available in the inner scope as long as you only read from it. As soon as you write to the variable, a local version is created, and you are no longer referring the variable from the outer scope. So as soon as var2 is set to 0, it becomes a local, or private variable. After the function declaration, a RuntimeManager object is created. Every parallel target gets passed this object even if it doesn’t get used, like in this example. But the RuntimeManager is used for anything that requires information from all the threads to be collected, like for a reduction, or for all the threads to have access to the same object likes locks. Finally, the submit function executes the target with the specified number of threads and then returns.

Source-to-source Translation 2

Here is another example. This time it’s a for directive inside of a parallel directive. At lines 6 and 7, the schedule and chunk variables are set to the values from the schedule clause. If the schedule clause isn’t included, then it will default to a static schedule with a chunk size of the number of iterations divided by the number of threads. In this if block, the main thread creates a manager object to feed the chunks to the threads according to the schedule. These three variables correspond to the three parameters given to the range function. if only 1 or 2 parameters are given, then the others are set to None, which is like null in C++. The setup function just initializes the manager with the range parameters. And then we get to the actual structure of the OpenMP for construct, which is an infinite loop, where each thread requests the next block of iterations to execute, and this continues until the done method of the manager returns true and the threads break out. Lastly, this line shows where the critical section starts by acquiring a lock, and then it ends here by releasing it.

Runtime API

Since there are only three functions in our runtime API, I put them all into one example. The omp\_get\_wtime() function is a wrapper around the time.time() function from the standard library. It returns the number of seconds from some arbitrary time in the past which is guaranteed not to change during the course of the program. So, the time we get back is always relative to the same time in the past. Here, each thread prints which thread it is, which is its id, and out of how many total threads that are executing the parallel region using the omp\_get\_thread\_num() and omp\_get\_num\_threads() functions. Here we get the time again, and then the last statement prints the time it took the code to execute by subtracting the start time from the end time.

Demo

I have two small programs I’m going to run. Let me switch to a terminal. First, as you can see, I’m in a folder in my user’s Desktop directory. The jython launcher has been added to the path and our modified Jython uses an environment variable called JYTHON\_HOME to know where the preprocessor should be launched from. So here is the code from the example on the last slide with the runtime functions. The only thing that’s different is the print statement is in a critical section now, which just keeps the output from getting jumbled by the threads printing at the same time. So, this alias will just run the preprocessor first, so you can see the transformed code. Okay, and now let’s run it. And here is the output. Let’s run it again and the output should be in a different order. So that shows that it is being executed by multiple threads in parallel. Okay, lets open it back up and put in an error. I’ll just add a simple indentation error. And now let’s run it again. The ANTLR parser catches a lot of the simple errors that people are likely to make when writing their code and the error messages are pretty helpful, it tells you the line number where the error happened and what the parser was expecting. So, we can fix it now. And run it again, and yeah, now we get the right output again.

So, the next program does three things. It creates a jarray and filles it with 20 million integers, they’re all just ones for now. And then it finds the sum with one thread here in this for loop. And then it does the same thing but in parallel with the parallel for directive and a reduction with number of threads set by a command line argument. So, the loop itself is exactly the same, the only difference is the OpenMP directive. And both of these are timed, and we will see which is faster. Let’s run it with 4 threads first. It takes a second to fill the array. And there we go, you can see the parallel version is faster and the results match, so that’s good. Let’s run it with 12 threads. So that took less time than 4 threads, but not a whole lot which makes sense.

Performance

Now let’s talk about performance. We tested the performance using two microbenchmarks based on simple parallel algorithms. All tests were run on a windows 10 machine with 6 cores and 12 threads. And each data point is the average of 20 runs. Also, all of the results were obtained using an OpenMP block even for the serial runs. The reason for this is that I ran the tests the first time with a separate serial code block, and the results were showing consistent superlinear speedup for small problem sizes. We suspect that this was because the main thread is doing some extra garbage collection or something like that and making the serial version look slower than it should be. So, running the tests through an OpenMP block makes sure they all get a new thread.

Microbenchmarks

The first microbenchmark we used was n by n matrix multiplication with row-wise partitioning. And the second was parallel sum. Both used a single jarray object to store the data, which was random integers in both cases. And both tests ran with 1, 2, 4, 8, and 12 threads.

Results

Here are the results. The numbers on the x axis are the problem size, so that’s 1 dimension of a matrix for matrix multiplication and then the number of integers that were added up for parallel sum. In terms of runtime, you can see that it takes less time to finish the more threads we use except for 8 and 12 threads on parallel sum, they went back and forth a bit. For the speedup, the pattern is interesting. Some of them started off high and then declined some and leveled off. And on the efficiency, we actually got higher than 1 for 2 and 4 threads on matrix multiplication and 2 threads on parallel sum at the beginning. This is going to require more investigation to know for sure what’s going on, but we suspect it has to do with the just in time compilation of jython. So, on the first run, it has to convert the target function to java bytecode and then all the runs from then on don’t have to do that, so it makes all the parallel runs look faster in relation to that first run then they are. But aside from that, these results are good enough to demonstrate parallelism, and that was our goal from the beginning.

Conclusions

In conclusion, we have implemented core OpenMP directives, clauses, and a runtime library in Python. We’ve made progress towards our design goals of having a usable, robust tool for teaching parallel concepts. We have tested the performance with two microbenchmarks and the results are adequate for our purposes. In the future, we plan to do extensive user testing to find any bugs and get user feedback. We also want to work with IDEs and see how the usability is and what improvements we can make there. We will further investigate our performance results and run more benchmarks. And we will add more functionality, like task parallelism, and we will bring more people on to the project and continue to improve it over time.