Parallelism is the idea of using multiple computer resources to speed up one unified process. This usually involves creating threads to use multiple processor cores in your program. Concurrency is then coordinating how to have multiple threads and processors work on a piece of data without causing any errors, redundancies, or mistakes. “Introducing Parallelism and Concurrency in the Data Structures Course” is an academic paper by Dan Grossman and Ruth E. Anderson that aims to convince both professors and students that adding Parallelism and Concurrency to the Data Structures curriculum is essential. There are a few reasons that they think this, firstly, they mention that as technology continues to advance, processors with multiple cores become ubiquitous and taking advantage of that improved hardware will result in more efficient programs. Second, learning about parallelism and concurrency in earlier classes will be beneficial when these topics are brought up in more advanced courses.

Other professors have also thought that adding these topics earlier in the curriculum is a good idea. However, there has not been an agreement between professors on where to add this and what topics to cover. So, in the second part of their paper, Grossman and Anderson suggest their curriculum. What they suggest is removing some specific implementations of certain data structures such as queues and stacks to make room for these new topics. Once some topics have been trimmed, you add their new curriculum. This curriculum includes: “Introduction to Multithreading”, “Basic Fork-Join Parallelism”, “Analyzing Parallel Programs”, ”More Parallel Algorithms”, ”Concurrency, Mutual Exclusion, Locks”, “Concurrency Programming Guidelines”, “Remaining Concurrency Topics”, and “Cross-Cutting Theme on Coverage”. “Introduction to Multithreading” would start the curriculum by having the students create multiple threads and either joining these threads or detaching these threads from each other to see how multiple threads work. Students would then begin to analyze how using multiple threads could improve the time complexity of certain algorithms. “Basic Fork-Join Parallelism” would have students use their new multithreading experience to apply it to certain simple algorithms. The specific example that they gave in the paper is a function that adds all of the values of an array. Using multithreading, you could have n processors work on array.size() / n values at a time which would significantly increase the speed of summing all of the values together. In “More Parallel Algorithms” students would dive deeper into how to quantify the increase in efficiency of a program that uses multithreading. In this section Grossman and Anderson discuss the idea of analyzing these multithread algorithms using work and span. Work is the time complexity that it would take for a singular processor to complete the function, and span is the time complexity it would take for the program to complete if there were an infinite number of processors. Then, the total Big O of an algorithm can be calculated with O((work/p) + span) where P is the number of processors.

In the “More Parallel Algorithms” section, they cover more algorithmic uses of multithreading (other than the summation of an array). Specifically, they mention using multithreading for maps and trees. “Concurrency, Mutual Exclusion, Locks”, in this subunit they discuss how using multiple threads can cause some issues when trying to access the same object/data/memory. For example, data races, which occurs when two threads try to edit a piece of data at the same time, but the order in which they edit that data is random, causing unreliable results. In “Concurrency Programming Guidelines” they discuss what rules you should follow so that you don’t run into those data issues. In “Remaining Concurrency Topics” we would brush over other important topics that we did not have time to fully dive into, such as deadlock and memory-model basics. In the last section “Cross-Cutting Themes on Coverage” they will mention topics that you could investigate on your own and that you will most likely see more of in other courses (such as scheduling).

In this last section of their paper, they discuss the data that they have collected from students that were in their classes that has this new curriculum. Using optional post-course surveys, they have discovered that when asked students about parallelism and concurrency, 35% of students said that it was the most important topic they covered and 68% of students said it was the most interesting topic of the course. Over a third of the students stating that a topic was the most important thing that they learned is very compelling evidence that these topics should be added to all data structures curriculum. Furthermore, having a topic that interests students could result in a more focused and well performing class. Another survey found that students that learned topics such as parallelism and concurrency in earlier computer science courses performed better in higher/more difficult courses in the future. Surveyed students said that their previous experience was either incredibly or fairly useful. Again, including these topics in their data structures courses has proven to be beneficial to computer science students in more difficult courses. So not only would adding these topics make you a more well-rounded programmer, but it would also improve your ability to learn in other classes.

Personally, I have done some mild multithreading programming in some personal projects but nothing that I would consider very impressive. Specifically, in the group lab that me and my lab group just submitted, I added some threads that act as doctor treating a patient. These threads then ran detached and would mark the doctor/room as available once treatment is complete. This feels to me like a very rudimentary use for threads, and I am not speeding up my program by doing this. I would love to learn how to properly use and apply them. I know I will probably see them in the future in more difficult classes where I will be expected to already know the basics of multithreading, which is another reason that I think the basics of it should be taught earlier in the curriculum. Or, at the very least, there should be a none core class that covers these topics. Overall, I think that the claim

Grossman and Anderson are making is a strong one and I would be happy if multithreading was in the curriculum for data structure courses.