Parallel Computing Primer

Parallel Computing is ubiquitous to modern computing. Whether people realize it or not, we are all using one form of parallel computing or another nearly any time we use a computer. Since it is so prevalent, some knowledge of this topic can be considered essential to the student in computer science. The following paragraphs discuss the types of parallel computing starting with a contrast of instruction level parallel computing vs task level parallel computing; after that there is a detailed overview of pipelining and superscaler execution. There are diagrams and examples where appropriate. From there the discussion moves to task parallelism and finishes with an example in python.

Instruction Level Parallelism vs Task Parallelism

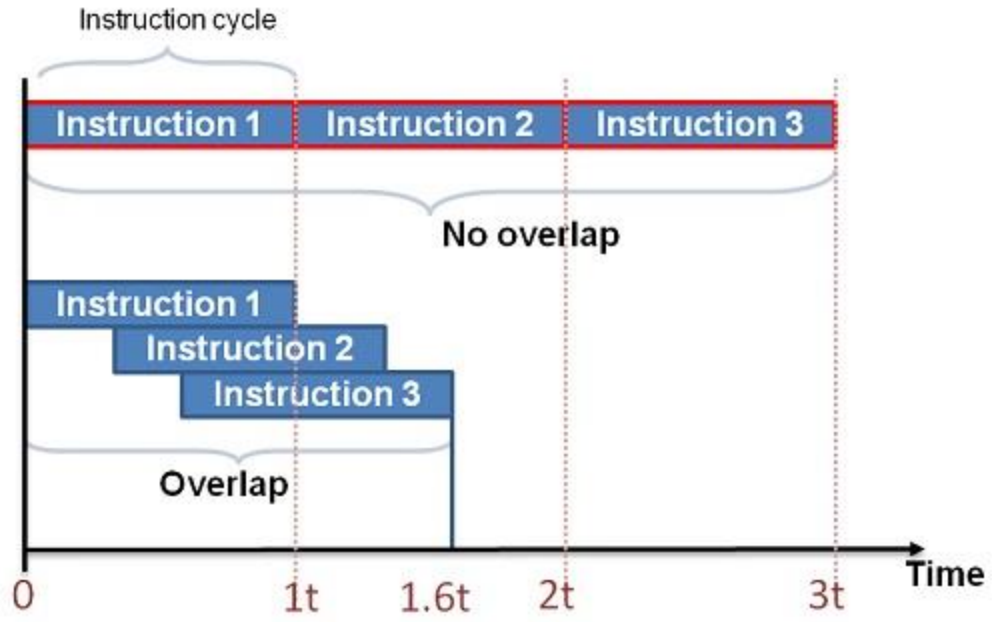
Instruction-level parallelism (ILP) and task parallelism are two distinct approaches to achieving parallelism in computing, focusing on different levels of granularity and execution strategy.

Instruction-level parallelism (ILP) focuses on maximizing the number of instructions executed concurrently within a single processor core. It involves techniques like pipelining, superscalar execution, and out-of-order execution to overlap instruction execution phases and utilize available resources more efficiently.

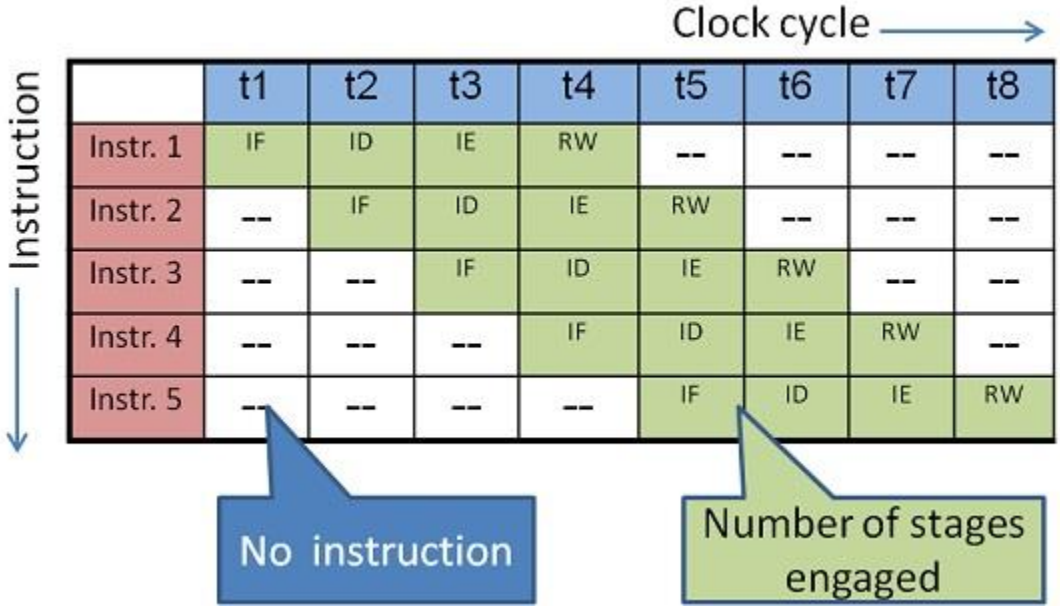
Pipelining

source (https://witscad.com/course/computer-architecture/chapter/concepts-of-pipelining)

Pipelining increases the performance of the system with simple design changes in the hardware. It facilitates parallelism in execution at the hardware level. Pipelining does not reduce the execution time of individual instructions but reduces the overall execution time required for a program. In a Pipelined design, multiple instructions are executed in a timing state, in an overlapped manner.



We have seen that an instruction cycle consists of a few machine cycles like Instruction Fetch, Decode, Execute, Result Writing, etc. A functional unit is designed to take care of each machine cycle operation. These functional units work independently but simultaneously with one of the instructions in the pipe. For example, during the instruction fetch machine cycle, the decode unit and execution unit are available for decoding and execution of another instruction; This design helps in achieving overlapped execution and called Pipelined Design.



- IF – Instruction fetch
- ID – Instruction decode
- IE – Instruction execute
- RW – Result writing

Although theoretically, all the machine cycles are identically timed, in practical implementation it is not perfectly balanced. Two examples may be given.

1. Instruction decode is simple static decoding and requires little time equal to the gate delay of the circuit involved
2. The time required for Memory operations are always decided by the memory for two reasons – memory is generally shared for access by CPU and IO subsystems; memory operations take more time than CPU operations. It is the memory that limits the CPU throttle. For these reasons memory cycles like instruction fetch and result writing in memory are likely to get extended timing.

It is important to note the performance. Refer to the phase diagram, in 8 clock cycles, 5 instructions have got executed in a four-stage pipelined design. The same would have taken 20 (5 instructions x 4 cycles for each) clock cycles in a non-pipelined architecture. The performance improvement depends on the number of stages in the design.

So, from an intuitive standpoint, we can define a clock cycles per instruction ratio.

ICT – Instuctions Completed per unit Time (measured in clock cycles)

$$ICT = \frac{clock\ cycles}{number\ of\ instuctions\ completed}$$

Referring to previous example, using pipelining, we get:

$$ICT = \frac{8}{5} = 1.6$$

Versus no pipelining

$$ICT = \frac{20}{4} = 4$$

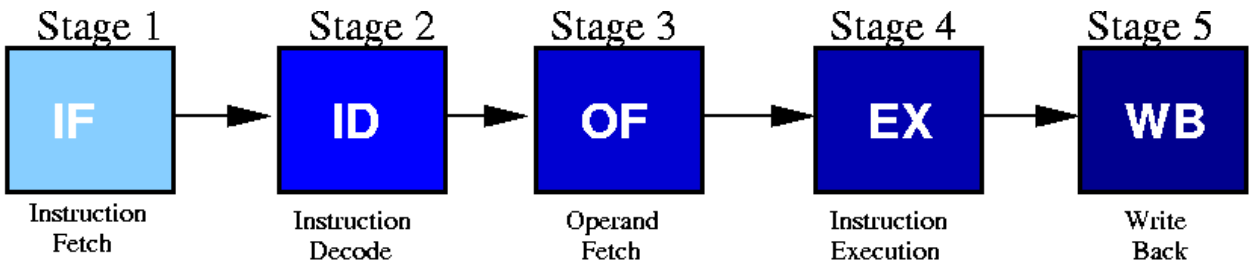
So, with pipelining, an instruction requires about 1.6 clock cycles to complete, compared to 4 clock cycles per instruction without the pipelining.

More information from:
https://users.cs.fiu.edu/~prabakar/cda4101/Common/notes/lecture03.html?ref=portfolio.jbm.fyi

In the above discussion, the pipeline described was a Four stage pipeline, with the stages being Instruction Fetch, Instruction Decode, Instruction Execute, and Result Writing. Other literature describes a Five Stage Pipeline in which the fetching of the operand is separated from the instruction decode stage.

This intuitively does not make sense because in order to decode an instruction, the first thing that needs to happen is getting the operand. Only after this is done the instructions various parts can be dealt with.

That being said, the following sections refer to a 5 stage pipeline.



	Time										
	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11
I1	IF	ID	OF	EX	WB						
I2		IF	ID	OF	EX	WB					
I3			IF	ID	OF	EX	WB				
I4				IF	ID	OF	EX	WB			
I5					IF	ID	OF	EX	WB		
I6						IF	ID	OF	EX	WB	
I7							IF	ID	OF	EX	WB

- Instruction **latency** is 5 cycles - Latency is a measure of how long an instruction takes to complete from start to finish.
- Ability to "issue" an instruction every cycle – Because we are pipelining the instructions (that is overlapping them in time) we can start a new instruction each clock cycle.
- After time 't4', an instruction finished every single cycle
- **Processor bandwidth** is one instruction per cycle

Pipeline Math

Example

- Assume each stage takes 20 nanoseconds.
- Without pipelining:
  - instruction latency is 100 ns
  - execute 10 instructions per microsecond, or 10 MIPS
- With pipelining:
  - cycle time is 20 nanosecods
  - execute 50 instructions per microsecond, or 50 MIPS.

Definitions:

- **nanosecond (ns)** - one billionth of a second (1 x 10^{-9} secs.)
- **microsecond (µs)** - one millionth of a second (1 x 10^{-6} secs.)
- **millisecond (ms)** - one thousanth of a second (1 x 10^{-3} secs.)

- **MIPS** - millions of instructions per second.

Pipeline Realities

There is an apparent factor of 5 speed-up in a 5-stage pipeline.

Problems:

- All stages must execute in the same amount of time, so "cycle" time must be as long as the stage that takes the maximal amount of time.
- Care must be put into decomposing the design into stages.
- Taking a 100 ns hardware operation and dividing into 5 parts that execute in exactly the same time is difficult.
- Idealistic calculations assume pipeline to always be "full".
- **Pipeline Hazards!**

Pipeline Hazards

- **Data Hazards** - when an instruction needs data that is not yet available. This is a data dependency.
- **Structural Hazards** - when the same hardware is needed by more than one instruction in the pipeline.
- **Control Hazards (Branch Hazards)** - when changes to the program counter affect the pipeline execution.

One thing to note in all of the above discussions is that interrupt stage has been left out of all of the examples. These examples are working with the idea that the cpu cycle is defined as fetch, decode, execute; the reality of modern computing is that the cpu cycle includes the interrupt stage at the end of the cpu cycle, meaning that the cpu cycle is fetch, decode, execute, interrupt.

Example of Instruction Level Parallelism

Source - https://www.geeksforgeeks.org/instruction-level-parallelism/

Suppose 4 operations can be carried out in a single clock cycle. So, there will be 4 functional units, each attached to one of the operations, branch unit, and common register file in the ILP execution hardware. The sub-operations that can be performed by the functional units are Integer ALU, Integer Multiplication, Floating Point Operations, Load, and Store. Let the respective latencies be 1, 2, 3, 2, 1.

Operations and Latencies	
Operation	Number of clock cycles required

Integer ALU (add, subtract, and, or)	1
Integer Multiplication	2
Floating Point ALU Operatoins	3
Load	2
Store	1

To be clear, there are 4 ALU units that can perform operations simultaneously. Each can perform operations during the same clock cycle. That is, unit 1 could be performing an Integer ALU operation (like an integer add) , while unit 2 could be performing Floating Point operation (like a floating point multiply). In this case, unit 1 would complete its operation 2 clock cycles before unit 2, because its operation takes only 1 clock cycle, and unit 2's operation takes 3 clock cycles.

Also, remember instructions can be executed out of order, as long as the inter-instruction dependencies are taken into account.

Let the sequence of instructions be:

1. y1 = x1*1010
2. y2 = x2*1100
3. z1 = y1+0010
4. z2 = y2+0101
5. t1 = t1+1
6. p = q*1000
7. clr = clr+0010
8. r = r+0001

Examining the instructions in more detail, we can see the operation types and the clock cycles required for the operation as follows in the table:

Cycle	Operation	Operation Type	Clock Cycles Required
1	y1 = x1 * 1010	Float Multiply	3
2	y2 = x2 * 1100	Float Multiply	3
3	z1 = y1 + 0010	Integer Add	1
4	Z2 = y2 + 0101	Integer Add	1
5	t1 = t1 + 1	Integer Add	1
6	p = q * 1000	Integer Multiply	2
7	clr = clr + 0010	Integer Add	1
8	r = r + 0001	Integer Add	1

So, if we run this sequence of instructions without pipelining (note this method is strictly sequential execution), we will see that we need 13 clock cycles to complete the instruction sequence. The yellow shaded regions indicate operations that require more than 1 clock cycle. The results are shown in the table below:

Cycle	Operation	Clock Cycles Required
1	y1 = x1 * 1010	3
2		
3		
4	y2 = x2 * 1100	3
5		
6		
7	z1 = y1 + 0010	1
8	Z2 = y2 + 0101	1
9	t1 = t1 + 1	1
10	p = q * 1000	2
11		
12	clr = clr + 0010	1
13	r = r + 0001	1

Now, if we run the same sequence with 4 functional units, running instructions simultaneously, and not in order when there are no dependencies, we can see that much time can be saved. Once again the yellow regions indicate continuing instructions, the light blue indicates a nop (no operation); the results are in the table below:

Cycle	Unit 1		Unit 2		Unit 3		Unit 4	
1	Integer Add	t1 = t1 + 1	Integer Add	clr = clr + 0010	Float Multiply	y1 = x1*1010	Float Multiply	y2 = x2*1100
2	Integer Add	r = r+0001	Integer Multiply	p = q*1000				
3	nop							
4	Integer Add	z1 = y1+0010	z1 = y2+0101					

- In cycle 1, units 1 and 2 handle out of order, non-dependency adds, while units 3 and 4 initiate floating point multiplication instructions.
- In cycle 2, unit 1 does another out of order, non-dependency add, while unit 2 initiates an integer multiply. Units 3 and 4 are still working on the multiplies that they started in cycle 1.
- In cycle 3, unit 1 does a nop because the only instructions remaining have dependencies that are still being processed by the other units. All other units work to complete their current instructions.
- In cycle 4, units 1 and 2, do integer add instructions with results from the previous instructions obtained from units 3 and 4.

In summary, the sequential processor requires 13 cycles to complete the sequence, while the pipelined processor uses 4.

Last but not least, an important consideration is what is the origin of these instruction arrangements? The most likely answer is the compiler/assembly process or the interpreter. In some compilers, when the optimization flags are set, the compiler can arrange to take advantage of pipelining.

Superscalar Execution vs Pipelining

Source: Google AI

Overview

Superscalar execution and pipelining are distinct but related techniques used to enhance processor performance, essentially aiming to execute multiple instructions concurrently.

Superscalar execution leverages multiple execution units to process instructions concurrently. It allows multiple instructions to be issued and executed simultaneously, even if they are at different stages of the pipeline.

Key Differences:

Focus:

Pipelining focuses on breaking down the execution process into stages, while superscalar execution focuses on utilizing multiple execution units.

Mechanism:

Pipelining overlaps instructions in a single execution unit, while superscalar execution uses multiple execution units to process multiple instructions simultaneously.

Performance Enhancement:

Both techniques aim to improve performance by increasing the number of instructions completed per unit time.

Superscalar Processing's Relationship to Pipelining

Superscalar processors often incorporate pipelining as well. A pipelined processor can be made superscalar by having multiple execution units within each pipeline stage.

Advantages of Superscalar Execution:

- Higher throughput: It can potentially execute more instructions per cycle than pipelining alone.
- Improved performance: Superscalar execution can lead to significant performance gains, especially for applications with inherent parallelism.

- Flexibility: Superscalar processors can be designed to execute different types of instructions simultaneously.

Advantages of Pipelining:

- Reduced instruction execution time: Pipelining can reduce the overall execution time of instructions by overlapping their processing stages.
- Increased throughput: It can increase the number of instructions completed per unit time.
- Simplified design: Pipelining can be implemented with less complex hardware than superscalar execution.

In essence, superscalar execution and pipelining are complementary techniques that can be combined to achieve significant performance improvements in modern processors.

Exercises

1. What is pipelining, and how is it different than normal sequential execution?
2. What are the typical stages of:
    a. A four stage pipleline?
    b. A five stage pipleline?
3. Where would interrupt handling fit into the pipeline strategy?
4. What is latency?
5. Given 5 instructions with a 4 stage pipleline, with each stage taking 10ns to execute, how long will it take the 5 instructions to execute?
6. Suppose the instructions from above are full of dependencies, meaning that pipelining cannot be used, how long will the instructions take to execute?
7. How is pipelining different from superscaler execution?
8. Can pipelining and superscaler execution be used at the same time?
9. How do programmers typically set a program up to take advantage of pipelining and/or superscaler execution?

Task Parallelism vs Instruction Level Parallelism (ILP)

Source: Google AI

Task parallelism, on the other hand, involves dividing a program or computation into independent tasks that can be executed concurrently on multiple processors or cores. This strategy focuses on exploiting the parallelism inherent in the problem itself, rather than within individual instructions.

Key Differences

- **Granularity:**

ILP works at the instruction level, focusing on how individual instructions are executed, while task parallelism deals with larger, independent units of work (tasks).

- **Scope:**

ILP is primarily concerned with maximizing the utilization of resources within a single processor, while task parallelism leverages multiple processors or cores to execute independent tasks.

- **Implementation:**

ILP is implemented through hardware and compiler techniques, while task parallelism requires explicit decomposition of the problem into independent tasks and their allocation to different processing units.

In essence, ILP focuses on squeezing more performance out of a single processor by overlapping instructions, while task parallelism distributes the workload across multiple processors to achieve faster overall execution.

Example

- **ILP**

Imagine a processor executing a loop with multiple additions. ILP techniques like pipelining can allow the processor to fetch and execute new instructions while previous instructions are still being processed.

- **Task parallelism**

Imagine a large image processing task. Task parallelism could involve dividing the image into smaller sections and assigning each section to a different processor for processing.

Task Parallelism

Source - Basic remarks on parallel computing: https://www.juanrga.com/2018/10/basic-remarks-on-parallel-computing.html

Task parallelism is the mode of parallelism where the tasks are divided among the processors to be executed simultaneously. Thread-level parallelism is when an application runs multiple threads at once.

For ordinary parallelization, a programmer or the compiler analyzes the instructions in a serial stream, finding control and data dependences, partitioning the original stream into almost independent sub streams (tasks), and inserting the necessary synchronization among tasks.

There are limits to the amount of parallelism
Not everything can be parallelized. Consider the quadratic equation and the elementary operations needed to solve it.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

n1 = b * b

n2 = 4 * a * c

n3 = n1 - n2

n4 = SQRT(n3)

n5 = -b + n4

n6 = -b - n4

n7 = 2 * a

x1 = n5 / n7

x2 = n6 / n7

Some of those operations are independent, but others are not. For instance, we cannot do the subtraction n3 without first knowing the values n1 and n2, and we cannot do the divisions x1 and x2 without first knowing the value of denominator n7. The maximum achievable parallelism will be

n1 = b * b; n2 = 4 * a * c; n7 = 2 * a

n3 = n1 - n2

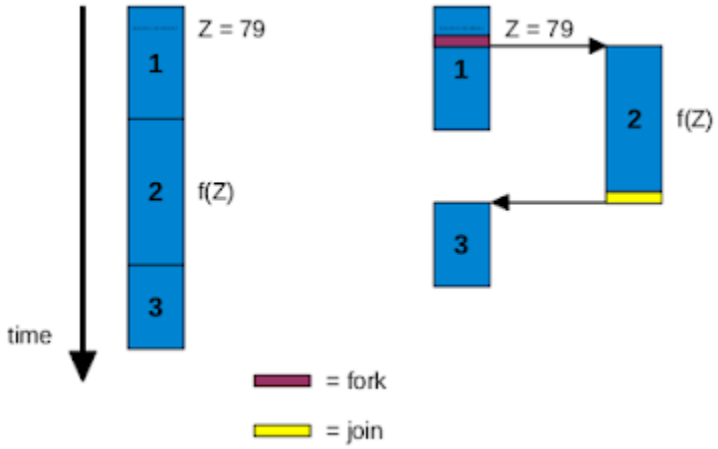
n4 = SQRT(n3)

n5 = -b + n4; n6 = -b - n4

x1 = n5 / n7; x2 = n6 / n7

The problem cannot be parallelized further due to *data dependences* among different operations.

Even for problems that can be parallelized, programmers have to confront hardware and software limits. For instance, consider the sequence of instructions illustrated in the left hand of next figure. The programmer has partitioned the code into an initialization subtask (1), a call to a procedure (2) that computes a function f that depends on variable Z, and a finalization subtask (3) that receives the value of the function. When executed linearly, the whole task takes a certain time to finish.



The three subtasks cannot start at the same time because the procedure depends on variables set by subtask (1), and the finalization subtask (3) depends on the value of f computed by (2). Some of would say that since we know the value of the variable Z, we could simply add that value to the beginning of the procedure and thus accelerate its execution. This can be done if the value of the variable is known at compile time, but if the value is only known at run-time --e.g. from user input or from data transmitted through the wire-- then the variable Z needs to be computed by the

initialization subtask before passing its value as parameter to the procedure.

To parallelize the execution of the procedure, extra code needs to be inserted in subtask (1) in the parent thread. This *fork* code creates a new thread and passes the needed parameters to the procedure. Once the procedure is executed, extra code added to the new thread takes the result of the evaluation of the function f and *joins* with the thread that called the procedure, to continue with execution of subtask 3.

The total time of execution is now smaller than in the original sequential execution. Parallelization has sped up execution. However, note that the code to execute now is larger because we have the fork and join sections. This is the overhead of the parallelization; it is extra code is not present in the original sequential algorithm, but it is code needed to synchronize and communicate the different threads in a parallel algorithm.

In the above example there is an initialization procedure and a subtask, there is only one procedure, and that fork and join overheads are small compared to the subtask's computation. However, things can be different and the cost of constructing and managing a thread can be greater than the computation time of the subtask itself; this happens for instance if the subtask is too small to compensate the overheads, in whose case the parallel algorithm can be actually slower than the serial algorithm.

The conclusion is that even for tasks that can be parallelized, the programmer has to evaluate the pros and cons and implement the optimal degree of parallelization. More parallel does not always imply faster!

Task Level Parallelism Example

For this example we will attempt to increase the execution efficiency of the bubble sort. Recall that the bubble sort is of time complexity order $O(n^2)$. Below is some pseudo code for the basic algorithm:

```
void bubbleMe(integer array numbers[])
{
        boolean swapping;
        integer j;

        swapping = true;
        while(swapping) {
                swapping = false;
                for(j=1;j< lengthOf(numbers[]), j++) {
                        if (numbers[j-1] > numbers[j]) {
                                swapping = true;
                                exchange(j-1, j, numbers[]);
                        }
                } /* End for. */
        } /* End while. */
}
```

So with a an order of $O(n^2)$ it would take about 1 million comparisons to sort a data set containing 1000 numbers. This makes sense, because 1000^2 is 1 million. We would like to take advantage of task parallelism to speed this up. It will be a 2-step process; first we will implement our speed up algorithm sequentially, then we will convert it to a parallel algorithm. For each version of the algorithm, we will measure the cpu time required to complete the sorting task. Hopefully the parallelization process will improve efficiency.

Our initial modifications to the code will be to divide the data set into two parts, run the bubble sort on each part separately, then merge the data sets into 1 sorted set. The bubble routine is modified as follows:

numList – array of unsorted numbers

startIndex – beginning of the area of the array to be sorted

endIndex – index of last array element to be sorted

```
bubblePartOfMe(numList, startIndex, endIndex):
    swapping = True
    count = 0
    while(swapping):
        swapping = False
        for j in range(startIndex+1, endIndex):
            count = count+1
            if (numList[j-1] > numList[j]):
                temp = numList[j-1]
                numList[j-1] = numList[j]
                numList[j] = temp
                swapping = True

    return numList, count
```

Our merge routine will be as the pseudo code below describes.

numList – array of unsorted numbers

a0 – starting index of the ‘a’ section of the array that is to be merged

aLen – length of ‘a’ section of the array

b0 – starting index of the ‘b’ section of the array that is to be merged

bLen – length of the ‘b’ section of the array

```
mergePartOfMe(numList, a0, aLen, b0, bLen):
    mSize = aLen + bLen
    a = a0
    b = b0
    scratch = []
    compares = 0
    for c in range(0, mSize):
        if ((a - a0) == aLen):
            scratch.append(numList[b])
            b = b+1
            compares = compares + 1
        elif ((b - b0) == bLen):
            scratch.append(numList[a])
            a = a+1
            compares = compares + 2
        elif (numList[a] < numList[b]):
            scratch.append(numList[a])
            a = a+1
            compares = compares + 3
        else:
            scratch.append(numList[b])
            b = b+1
            compares = compares + 3

    for c in range(0, mSize):
        numList[c] = scratch[c]

    return numList, compares
```

Note: The above pseudo code can be optimized; it is written this way illustrate the concepts of merging.

If the random set of numbers is sorted using the basic bubble sort the number of comparisons is about 921,000. If the same data set is separated into 5 different pieces, bubbled, and then merged the comparison count, bubble comparisons + merge comparisons, is between 188,000 and 189,000 compares. This is quite an improvement in efficiency, at least based on the number of comparisons. How did this happen, given both algorithms are $O(n^2)$?

The explanation comes down to the arithmetic based on the squaring of numbers. The following equations illustrate what is occurring.

Comparisons needed to sort the original data set, squared because the algorithm is $O(n^2)$

$1000^2 = 1{,}000{,}000$

Breaking the data set into 5 equally sized pieces….

200 + 200 + 200 + 200 + 200 = 5 * 200 = 1000

Comparisons needed to sort a piece of the data set using bubble....

200^2 = 40,000

Total sort comparisons needed to sort the 5 equally sized pieces of the data set....

5*40,000 = 200,000

Total comparisons = bubble comparisons + merge comparisons

Total comparisons for our data set is about 188,500 comparisons, with about 8,000 comparisons being used for merging the data. This is somewhat less than our estimate from above because there are duplicates in the data set, and the swapping variable adds a bit of efficiency to the bubble sort.

As for cpu time used, implemented in python, the non-optimized code reported:

```
cpu time used =  0.109375
```

The optimized code reported:

```
total time needed for bubble and merge =  0.03125
```

This is a speed up by a factor of 3.5 times. This is pretty good. It is not as good as 5 times as may be expected due to dividing the data set into 5 pieces, but still, all said and done, it is still good. The next question is, can running the bubble in sort in parallel increase the efficiency in a similar manner.

For this exercise we will use python. Google AI gives the following advice:

Choosing the Right Approach:

CPU-bound tasks:

If your tasks are computationally intensive, use multiprocessing or concurrent.futures.ProcessPoolExecutor to leverage multiple CPU cores.

I/O-bound tasks:

If your tasks involve a lot of waiting for I/O operations (like network requests), use asyncio to achieve concurrency without blocking the main thread.

Working with python and the multiprocessing library the following python code was developed:

```
import multiprocessing
import time
from multiprocessing import Pool, freeze_support

def loadArrayFromFile(fName):
    print("reading ", fName)

    lun = open(fName, "r")

    reading = True

    # Create an emptyh array
    arr = []
    while reading :
        x = lun.readline()
        # Check to see if we are at the end of the file.
        if (x) :
            arr.append(int(x))
        else :
            reading = False

    lun.close()

    return arr


def showArray(l):
    for x in l:
        print(x)
    return


def bubblePartOfMe(numList, startIndex, endIndex):
    swapping = True
    count = 0
    while(swapping):
        swapping = False
        for j in range(startIndex+1, endIndex):
            count = count+1
            if (numList[j-1] > numList[j]):
                temp = numList[j-1]
                numList[j-1] = numList[j]
                numList[j] = temp
                swapping = True

    return numList, count


def mergePartOfMe2(numListA, a0, aLen, numListB, b0, bLen):
    mSize = aLen + bLen
    a = a0
    b = b0
    scratch = []
    compares = 0
    for c in range(0, mSize):
        if ((a - a0) == aLen):
            scratch.append(numListB[b])
            b = b+1
            compares = compares + 1
        elif ((b - b0) == bLen):
            scratch.append(numListA[a])
            a = a+1
            compares = compares + 2
        elif (numListA[a] < numListB[b]):
```

```
            scratch.append(numListA[a])
            a = a+1
            compares = compares + 3
        else:
            scratch.append(numListB[b])
            b = b+1
            compares = compares + 3

    #for c in range(0, mSize):
        #numList[c] = scratch[c]

    #return numList, compares
    return scratch, compares



def showPartOfArray(numList, start, end):
    for j in range(start, end):
        print(numList[j], end = " ")


if __name__ == '__main__':
    # Load the data file.
    l = loadArrayFromFile("numbers.txt")
    # Set up the multiprocessing environment.
    pool = Pool(multiprocessing.cpu_count())

    # Set up the two tasks to run in parallel, no blocking.
    result1 = pool.apply_async(bubblePartOfMe, [l, 0, 500])
    result2 = pool.apply_async(bubblePartOfMe, [l, 500, 1000])
    freeze_support()

    # Get start time, and kick off the processes.
    startTime = time.time()
    l0, x0 = result1.get(timeout=10)
    l1, x1 = result2.get(timeout=10)
    endTime = time.time()
    sortTime = endTime - startTime

    # Show the results of the sorting.
    print("l0, l1 arrays:")
    showPartOfArray(l0, 0, 500)
    showPartOfArray(l1, 500, 1000)

    # Merge the two arrays.
    print("\nsizes of l0, and l1 ", len(l0), len(l1))
    startTime = time.time()
    l, mergeCompares = mergePartOfMe2(l0, 0, 500, l1, 500, 500)
    endTime = time.time()
    mergeTime = endTime - startTime


    # Show the merged results.
    print("l = ...")
    print(l)

    totalTime = sortTime + mergeTime
    print("totalTime = ", totalTime)
```

In this code, the data set is broken into 2 sections, each 500 items, each are sorted in parallel, then merged. This process took about 0.2 seconds of cpu time. Notice, this is longer than the

sequential process earlier, and also longer than running the algorithm on the whole data set. Factors that could be in play include:

- Splitting the data set into 2 pieces verses 5 takes quite a bit more comparisons, so it should not be as fast.
- It should be as fast as not doing the parallel processing, but the overhead may be a factor here.
- The data set size may be two small for the parallel processing to mitigate the overhead involved in setting it up.

A Sample for Assignment and Test

1. Using the examples and pseudo code in the previous sections as a guide, create a sort that uses the bubble algorithm and merge algorithm to sort a file of random numbers that you create.
    a. Let the file have 10,0000 numbers ranging from 0 to 1000.
    b. Find the number of comparisons and the cpu time for the following data arrangements:
        i. No separate data sets, sort all 10,000 numbers using bubble sort.
        ii. Bubble the first 5000 numbers and the second 5000 numbers separately, then merge the sets.
        iii. Do the same as the previous item except use 10 partitions of 1000 numbers . Sort and merge them.
    c. Now, using the python multiprocessing library, and the example in the section above, repeat item ii and iii from item b above.
    d. Answer the following questions:
        i. Why was dividing the data set into smaller pieces and merging faster?
        ii. Was using the parallel technique quicker than not using it? Give reasons to support your answer.