1. Let's think about parallelism, but with real-world examples.

a) What "major" tasks/steps exist in building a house? What tasks can be done in parallel assuming we have "unlimited" workers?

b) Putting together a 5,000-piece jiggle-saw puzzle with different number of people:

- 2 people
- 5 people
- 25 people
- 100 people

c) Preparing food for a large (1,000 people) banquet with salad, entree, side-dish, dessert.

2. What are the main motivation(s) of writing a parallel program?

3. Two types of parallelism:
- task parallelism - split program into major tasks and solve as many tasks in parallel as possible
- data parallelism - partition the data across the processors with each processor doing the same type of calculations on their own chuck of data

a) Which of the above approaches is more scalable (can utilize more processors)?

b) How might a combination of the two be used?

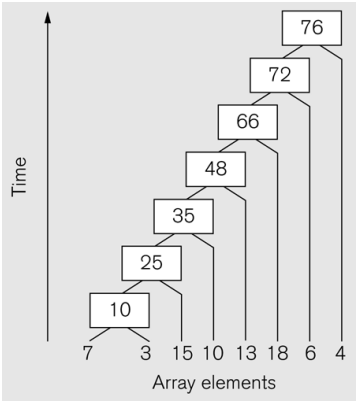
4. There is a "paradigm shift" making parallel programming conceptually different from sequential programming. A simple example is summing an array x containing n elements.

Sequential Algorithm:

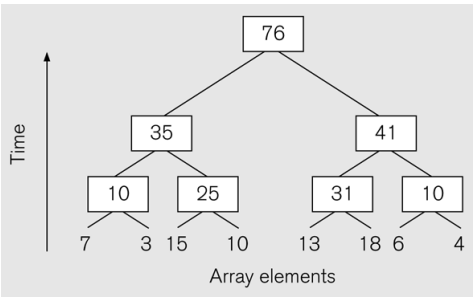
```
sum = 0;
for (i = 0; i < n; i++) {
    sum = sum + x[i];
} // end for
```

a) How long would each algorithm take?

b) How many processors does the pair-wise summation algorithm utilize?



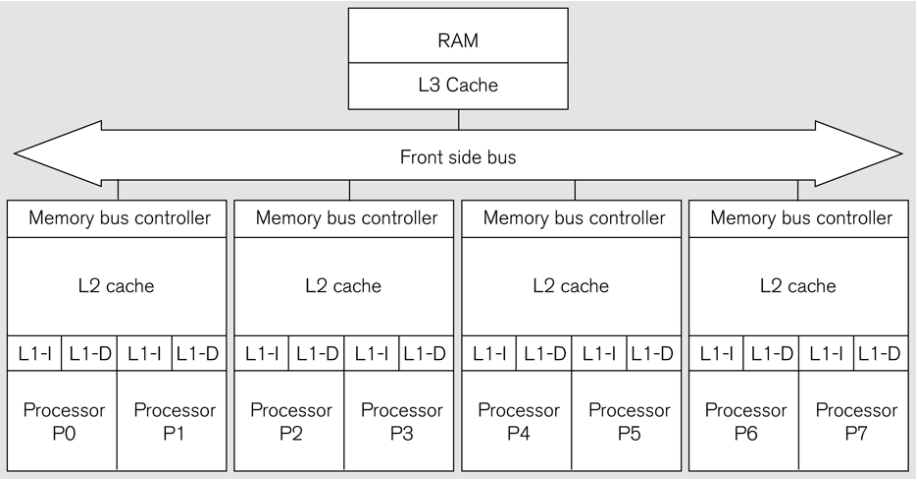
Parallel Pair-Wise Summation Algorithm



General steps for designing parallel programs suggested by Ian Foster ("*Foster's methodology*"):
1. *Partitioning*: Divide computation and the data operated on into small tasks. Here the focus is identifying tasks that can be executed in parallel. (We might have true (RAW) data dependencies to consider.)
2. *Communication*: Determine what communication is needed to be carried out among the tasks identifed in step 1
3. *Agglomeration or aggregation*: Combine tasks and communications identified in the previous steps into larger tasks. For example, if task A must be executed before task B can be executed, it may make sense to aggregate them into a single composite task.
4. *Mapping*: Assign the composite tasks identified in the previous step to processes/threads. This should be done so that communication is minimized, and each process/thread get roughly the same amount of work.

Let's uses the summing an array problem to illustrate the complexity of parallel programming. Details:

- Problem: sum the numbers in an array
- Parallel computer used: 8-cores
- Multitheaded program
  - all threads share global memory space: program, array, and sum
  - each thread has its own PC (pgm. counter) and run-time stack



Try to apply Foster's method on an array in RAM:

0	1	2	3	4	5	6	7	
20	35	15	10	50	30	5	40	. . . .

Matrix Multiplication is a frequently used operation that takes two matrices A (m x q) and matrix B (q x n) and produces matrix C (m x n), where c_{ij} is the dot product of the i^{th} row of A with the j^{th} column of B. In other words,

$$c_{ij} = \sum_{k=0}^{q-1} a_{ik} * b_{kj}$$

For example: A × B = C

$$\begin{pmatrix} 2 & 3 & 1 \\ 0 & 2 & 1 \\ 2 & 2 & 1 \\ 0 & 3 & 2 \end{pmatrix} \times \begin{pmatrix} 2 & 2 \\ 1 & 0 \\ 2 & 1 \end{pmatrix} = \begin{pmatrix} 9 & 5 \\ 4 & 1 \\ 8 & 5 \\ 7 & 2 \end{pmatrix}$$

The sequential code for matrix multiplication is:

```
for i = 0 to m-1 do
    for j = 0 to n-1 do
        cij = 0
            for k = 0 to q-1 do
                cij = cij + aik * bkj
            end for k
     end for j
end for i
```

Your assignment is to implement matrix multiplication serially using C/C++ on student.cs.uni.edu to compile and time your code.