Pipelined Computations

1. Functional Decomposition
   - car manufacturing with three plants
   - speedup for $n$ inputs in a $p$-stage pipeline
   - loop unrolling

2. Pipeline Implementations
   - processors in a ring topology
   - pipelined addition
   - pipelined addition with MPI
Functional Decomposition
- car manufacturing with three plants
- speedup for \( n \) inputs in a \( p \)-stage pipeline
- loop unrolling

Pipeline Implementations
- processors in a ring topology
- pipelined addition
- pipelined addition with MPI
Consider a simplified car manufacturing process in three stages: (1) assemble exterior, (2) fix interior, and (3) paint and finish:

The corresponding *space-time diagram* is below:

After 3 time units, one car per time unit is completed.
denoising a signal

Every second we take 256 samples of a signal:

- \( P_1 \): apply FFT,
- \( P_2 \): remove low amplitudes, and
- \( P_3 \): inverse FFT.

An alternative space-time diagram is below:

Observe: the consumption of a signal is sequential.
**p-stage pipelines**

A pipeline with $p$ processors is a *p-stage pipeline*.

Suppose every process takes one time unit to complete.

How long till a $p$-stage pipeline completes $n$ inputs?

A $p$-stage pipeline on $n$ inputs:

- After $p$ time units the first input is done.
- Then, for the remaining $n - 1$ items, the pipeline completes at a rate of one item per time unit.

$\Rightarrow p + n - 1$ time units for the $p$-stage pipeline to complete $n$ inputs.

A time unit is called a *pipeline cycle*.

The time taken by the first $p - 1$ cycles is the *pipeline latency*. 

Pipelined Computations

1. Functional Decomposition
   - car manufacturing with three plants
   - speedup for $n$ inputs in a $p$-stage pipeline
   - loop unrolling

2. Pipeline Implementations
   - processors in a ring topology
   - pipelined addition
   - pipelined addition with MPI
speedup

Consider \( n \) inputs for a \( p \)-stage pipeline:

\[
S(p) = \frac{n \times p}{p + n - 1}.
\]

For fixed number \( p \) of processors:

\[
\lim_{n \to \infty} \frac{p \times n}{n + p - 1} = p.
\]

Pipelining speeds up multiple sequences of heterogeneous jobs.

- Pipelining is a functional decomposition method to develop parallel programs.
- Recall the classification of Flynn: MISD = Multiple Instruction Single Data stream.
floating-point addition

A floating-point number consists of a sign bit, an exponent and a fraction (or mantissa):

$$\pm e \text{ (8 bits)} \quad f \text{ (23 bits)}$$

Floating-point addition could be done in 6 cycles:

1. unpack fractions and exponents
2. compare exponents
3. align fractions
4. add fractions
5. normalize result
6. pack fraction and exponent of result

Adding two vectors of $n$ floats with 6-stage pipeline takes $n + 6 - 1$ pipeline cycles, instead of $6n$ cycles.

$\Rightarrow$ Capable of performing one flop per clock cycle.
Figure 2-3. The Intel Core Microarchitecture Pipeline Functionality
Pipelined Computations

1. Functional Decomposition
   - car manufacturing with three plants
   - speedup for $n$ inputs in a $p$-stage pipeline
   - loop unrolling

2. Pipeline Implementations
   - processors in a ring topology
   - pipelined addition
   - pipelined addition with MPI
The Leibniz series

\[
\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots
\]

converges very slowly.

This example is based on section 3.2.2 on loop unrolling in *Scientific Programming and Computer Architecture* by Divakar Viswanath, Springer-Verlag, 2017.

The above reference offers a very detailed explanation.

We can already illustrate the main point in Julia.
a straightforward implementation

The branching in the straightforward code below prevents a pipelined execution of the floating-point operations.

```plaintext
function leibniz1(N::Int)
    s = 1.0
    for i=1:N
        if(i%2 == 1)
            s = s - 1.0/(2.0*i + 1.0)
        else
            s = s + 1.0/(2.0*i + 1.0)
        end
    end
    return s
end
```

\[
\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots
\]
Summing the even and odd terms separately avoids branching, allows a pipelined executions of the floating-point operations.

```markdown
function leibniz2(N::Int)
    s = 1.0
    for i=2:2:N
        s = s + 1.0/(2.0*i + 1.0)
    end
    for i=1:2:N
        s = s - 1.0/(2.0*i + 1.0)
    end
    return s
end
```

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots = 1 + \frac{1}{5} + \frac{1}{9} + \cdots - \frac{1}{3} - \frac{1}{7} - \frac{1}{11} - \cdots$$
benchmarking

using BenchmarkTools

println(4.0*leibniz1(10^8))
@btime leibniz1(10^8)

println(4.0*leibniz2(10^8))
@btime leibniz2(10^8)

with output:

3.141592663589326
  239.600 ms (0 allocations: 0 bytes)
3.1415926635801443
  125.266 ms (0 allocations: 0 bytes)

Julia 1.8.5 on pascal:
  • two 22-core Intel Xeon E5-2699v4 Broadwell at 2.20GHz,
  • 256GB of internal memory at 2400MHz.
Pipelined Computations

1. Functional Decomposition
   - car manufacturing with three plants
   - speedup for $n$ inputs in a $p$-stage pipeline
   - loop unrolling

2. Pipeline Implementations
   - processors in a ring topology
   - pipelined addition
   - pipelined addition with MPI
processors in a ring topology

A ring topology is a natural way to implement a pipeline.

A manager/worker organization:

- Node 0 receives input and sends to node 1.
- Every node \( i \), for \( i = 1, 2, \ldots, p - 1 \):
  1. receives an item from node \( i - 1 \),
  2. performs operations on the item,
  3. sends processed item to node \( (i + 1) \mod p \).

At the end of one cycle, node 0 has the output.
one pipeline cycle with MPI

$ mpirun -np 4 ./pipe_ring
One pipeline cycle for repeated doubling.
Reading a number...
2
Node 0 sends 2 to the pipe...
Processor 1 receives 2 from node 0.
Processor 2 receives 4 from node 1.
Processor 3 receives 8 from node 2.
Node 0 received 16.
$

This example is a type 1 pipeline
efficient only if we have more than one instance to compute.
space time diagrams for type 2 and type 3 pipelines

Type 2:

Type 3:
void manager ( int p )
/*
 * The manager prompts the user for a number
 * and passes this number to node 1 for doubling.
 * The manager receives from node p-1 the result. */
{
    int n;
    MPI_Status status;

    printf("One pipeline cycle for repeated doubling.\n");
    printf("Reading a number...\n");
    scanf("%d",&n);
    printf("Node 0 sends %d to the pipe...\n",n);
    fflush(stdout);
    MPI_Send(&n,1,MPI_INT,1,tag,MPI_COMM_WORLD);
    MPI_Recv(&n,1,MPI_INT,p-1,tag,MPI_COMM_WORLD,&status);
    printf("Node 0 received %d.\n",n);
}
MPI code for the workers

```c
void worker ( int p, int i )
/*
* Worker with identification label i of p
* receives a number,
* doubles it and sends it to node i+1 mod p. */
{
    int n;
    MPI_Status status;

    MPI_Recv(&n,1,MPI_INT,i-1,tag,MPI_COMM_WORLD,&status);
    printf("Processor %d receives %d from node %d.\n", i,n,i-1);
    fflush(stdout);
    n *= 2; /* double the number */
    if(i < p-1)
        MPI_Send(&n,1,MPI_INT,i+1,tag,MPI_COMM_WORLD);
    else
        MPI_Send(&n,1,MPI_INT,0,tag,MPI_COMM_WORLD);
}
```
Pipelined Computations

1. Functional Decomposition
   - car manufacturing with three plants
   - speedup for $n$ inputs in a $p$-stage pipeline
   - loop unrolling

2. Pipeline Implementations
   - processors in a ring topology
   - pipelined addition
   - pipelined addition with MPI
pipelined addition

Consider 4 processors in a ring topology:

\[
\begin{array}{c}
P_0 \rightarrow P_1 \rightarrow P_2 \rightarrow P_3 \\
\end{array}
\]

To add a sequence of 32 numbers, with data partitioning:

\[
\begin{align*}
A_k &= \sum_{j=0}^{k} a_j \\
B_k &= \sum_{j=0}^{k} b_j \\
C_k &= \sum_{j=0}^{k} c_j \\
D_k &= \sum_{j=0}^{k} d_j
\end{align*}
\]

The final sum is \( S = A_7 + B_7 + C_7 + D_7 \).
space-time diagram for pipelined addition

\[ a_0, a_1, \ldots, a_7, b_0, b_1, \ldots, b_7, c_0, c_1, \ldots, c_7, d_0, d_1, \ldots, d_7. \]

\[ A_k = \sum_{j=0}^{k} a_j \quad B_k = \sum_{j=0}^{k} b_j \quad C_k = \sum_{j=0}^{k} c_j \quad D_k = \sum_{j=0}^{k} d_j \]

Denote \( S_1 = A_7 + B_7, \ S_2 = S_1 + C_7, \ S = S_2 + D_7. \)
speedup for pipelined addition

We finished addition of 32 numbers in 12 cycles: $12 = 32/4 + 4$.

In general, with $p$-stage pipeline to add $n$ numbers:

$$S(p) = \frac{n - 1}{n/p + p}$$

For fixed $p$: $\lim_{n \to \infty} S(p) = p$. 
Pipelined Computations

1. Functional Decomposition
   - car manufacturing with three plants
   - speedup for $n$ inputs in a $p$-stage pipeline
   - loop unrolling

2. Pipeline Implementations
   - processors in a ring topology
   - pipelined addition
   - pipelined addition with MPI
using 5-stage pipeline

$ mpirun -np 5 ./pipe_sum
The data to sum:  1  2  3  4  5  6  7  8  9 10 11 12 13 14  \ 
15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
Manager starts pipeline for sequence 0...
Processor 1 receives sequence 0:  3  3  4  5  6
Processor 2 receives sequence 0:  6  4  5  6
Processor 3 receives sequence 0:  10  5  6
Processor 4 receives sequence 0:  15  6
Manager received sum 21.
Manager starts pipeline for sequence 1...
Processor 1 receives sequence 1:  15  9 10 11 12
Processor 2 receives sequence 1:  24 10 11 12
Processor 3 receives sequence 1:  34 11 12
Processor 4 receives sequence 1:  45 12
Manager received sum 57.
Manager starts pipeline for sequence 2...
Processor 1 receives sequence 2 : 27 15 16 17 18
Processor 2 receives sequence 2 : 42 16 17 18
Processor 3 receives sequence 2 : 58 17 18
Processor 4 receives sequence 2 : 75 18
Manager received sum 93.
Manager starts pipeline for sequence 3...
Processor 1 receives sequence 3 : 39 21 22 23 24
Processor 2 receives sequence 3 : 60 22 23 24
Processor 3 receives sequence 3 : 82 23 24
Processor 4 receives sequence 3 : 105 24
Manager received sum 129.
Manager starts pipeline for sequence 4...
Processor 1 receives sequence 4 :  51 27 28 29 30
Processor 2 receives sequence 4 :  78 28 29 30
Processor 3 receives sequence 4 :  106 29 30
Processor 4 receives sequence 4 :  135 30
Manager received sum 165.
The total sum : 465
$
MPI code

```c
void pipeline_sum ( int i, int p )
/* performs a pipeline sum of p*(p+1) numbers */
{
    int n[p][p-i+1];
    int j,k;
    MPI_Status status;

    if(i==0) /* manager generates numbers */
    {
        for(j=0; j<p; j++)
            for(k=0; k<p+1; k++)
                n[j][k] = (p+1)*j+k+1;
        if(v>0)
            {
                printf("The data to sum : ");
                for(j=0; j<p; j++)
                    for(k=0; k<p+1; k++)
                        printf(" %d",n[j][k]);
                printf("\n");
            }
    }
}
```
loop for manager

for (j=0; j<p; j++)
    if (i==0) /* manager starts pipeline of j-th sequence */
    {
        n[j][1] += n[j][0];
        printf("Manager starts pipeline for sequence %d...
", j);
        MPI_Send(&n[j][1], p, MPI_INT, 1, tag, MPI_COMM_WORLD);
        MPI_Recv(&n[j][0], 1, MPI_INT, p-1, tag, MPI_COMM_WORLD, &status);
        printf("Manager received sum %d.\n", n[j][0]);
    }
else /* worker i receives p-i+1 numbers */
else /* worker i receives p-i+1 numbers */
{
    MPI_Recv(&n[j][0],p-i+1,MPI_INT,i-1,tag,
             MPI_COMM_WORLD,&status);
    printf("Processor %d receives sequence %d : ",i,j);
    for(k=0; k<p-i+1; k++) printf(" %d", n[j][k]);
    printf("\n");
    n[j][1] += n[j][0];
    if(i < p-1)
        MPI_Send(&n[j][1],p-i,MPI_INT,i+1,tag,
                 MPI_COMM_WORLD);
    else
        MPI_Send(&n[j][1],1,MPI_INT,0,tag,MPI_COMM_WORLD);
}

if(i==0) /* manager computes the total sum */
{
    for(j=1; j<p; j++) n[0][0] += n[j][0];
    printf("The total sum : %d\n",n[0][0]);
}
We started chapter 5 in the book of Wilkinson and Allen.

Exercises:

1. Describe the application of pipelining technique for grading $n$ copies of an exam that has $p$ questions. Explain the stages and make a space-time diagram.

2. Write code to use the 4-stage pipeline to double numbers for a sequence of 10 consecutive numbers starting at 2.

3. Consider the evaluation of a polynomial $f(x)$ of degree $d$ given by its coefficient vector $(a_0, a_1, a_2, \ldots, a_d)$, using Horner’s method, e.g., for $d = 4$: $f(x) = (((a_4 x + a_3)x + a_2)x + a_1)x + a_0$. Give MPI code of this algorithm to evaluate $f$ at a sequence of $n$ values for $x$ by a $p$-stage pipeline.