# Software Pipelining

# Pipelining versus Parallel

Suppose it takes 3 units of time to assemble a widget. Furthermore, suppose the assembly consists of three steps– A, B, and C –and each step requires exactly one unit of time.

- **Sequential**. Time to make $n$ widgets is $3n$, with one widget per 3 units of time.

# Pipelining versus Parallel

Suppose it takes 3 units of time to assemble a widget. Furthermore, suppose the assembly consists of three steps– A, B, and C –and each step requires exactly one unit of time.

- **Sequential**. Time to make $n$ widgets is $3n$, with one widget per 3 units of time.
- **Pipelined**. Each of the three steps have been assigned to a separate machine. Then the pipeline produces the first widget in three time units, but after that one new widget appears every time unit. Time needed to assemble $n$ widgets is $n+2$.

# Pipelining versus Parallel

Suppose it takes 3 units of time to assemble a widget. Furthermore, suppose the assembly consists of three steps– A, B, and C –and each step requires exactly one unit of time.

- **Sequential**. Time to make $n$ widgets is $3n$, with one widget per 3 units of time.
- **Pipelined**. Each of the three steps have been assigned to a separate machine. Then the pipeline produces the first widget in three time units, but after that one new widget appears every time unit. Time needed to assemble $n$ widgets is $n+2$.
- **Parallel**. Three independent widget assemblers. Produces three widgets in every three time units. Time to assemble $n$ widgets is $3\lceil n/3 \rceil$ time units.

# Pipelining

In general, suppose we have $p$ sub-assembly tasks, each taking one time unit.

- First widget comes out in $p$ time units.
- After that one widget comes out every time unit.

Therefore $n$ widgets require $n-1+p$ time units. Sequentially, $n$ widgets require $np$ time units. The speedup is:

$$S_p(n) = \frac{np}{n-1+p} = \frac{p}{1-\frac{1-p}{n}}$$

Pipelining is widely used in hardware design. Here we will see its use in software. Such use is termed **software pipelining**.

# Example 1: Prime Number Generation

*Sieve of Eratosthenes* is an efficient sequential algorithm for generating all primes in the range $2 \ldots n$.

- Start with a list of $n$ numbers $2, 3, \ldots, n$.
- At iteration $i$, strike out all multiples of the current prime number, which is always at the front of the remaining list. We need to iterate at most $\sqrt{n}$ times.

| iteration | ~numbers eliminated |
|---|---|
| 1 | $\frac{n}{2}$ |
| 2 | $\frac{n}{2\times 3}$ |
| 3 | $\frac{n}{2\times 3\times 5}$ |
| 4 | $\frac{n}{2\times 3\times 5\times 7}$ |
| $\ldots$ | $\ldots$ |

# Sieve of Eratosthenes (contd.)

The **Prime Number Theorem** states that we have $\Theta(n/\lg n)$ prime numbers between 1 and $n$. So we will, in general, stop the sieve iterations when the count of the remaining numbers is $\Theta(n/\lg n)$. In each iteration, we are eliminating a constant fraction of the numbers. So the number of iterations is $O(\lg \lg n)$. The sequential runtime is $T^*(n) = O(n \lg \lg n)$.
Some improvements to be used in an actual sieve implementation.

- Eliminate all even numbers (except 2) from the starting list.

# Sieve of Eratosthenes (contd.)

The **Prime Number Theorem** states that we have $\Theta(n/\lg n)$ prime numbers between 1 and $n$. So we will, in general, stop the sieve iterations when the count of the remaining numbers is $\Theta(n/\lg n)$. In each iteration, we are eliminating a constant fraction of the numbers. So the number of iterations is $O(\lg\lg n)$. The sequential runtime is $T^*(n) = O(n \lg\lg n)$.
Some improvements to be used in an actual sieve implementation.

- Eliminate all even numbers (except 2) from the starting list.
- Eliminate all numbers that are not a multiple of 2 or 3 (except 2 and 3) from the starting list. So the list look like:

$$2, 3, 5, 7, 11, 13, 17, 19, 23, 25, \ldots$$

# Sieve of Eratosthenes (contd.)

The **Prime Number Theorem** states that we have $\Theta(n/\lg n)$ prime numbers between 1 and $n$. So we will, in general, stop the sieve iterations when the count of the remaining numbers is $\Theta(n/\lg n)$. In each iteration, we are eliminating a constant fraction of the numbers. So the number of iterations is $O(\lg\lg n)$. The sequential runtime is $T^*(n) = O(n\lg\lg n)$.
Some improvements to be used in an actual sieve implementation.

- Eliminate all even numbers (except 2) from the starting list.
- Eliminate all numbers that are not a multiple of 2 or 3 (except 2 and 3) from the starting list. So the list look like:

  $$2, 3, 5, 7, 11, 13, 17, 19, 23, 25, \ldots$$

  This list can be generated with $3i+2, 3i+4$, where $i = 1, 3, 5, 7, \ldots$.
- We can further eliminate all numbers of the form $30m \pm 5, m \geq 1$, thereby eliminating all spurious multiples of 5.

# Sieve of Eratosthenes (contd.)

The **Prime Number Theorem** states that we have $\Theta(n/\lg n)$ prime numbers between 1 and $n$. So we will, in general, stop the sieve iterations when the count of the remaining numbers is $\Theta(n/\lg n)$. In each iteration, we are eliminating a constant fraction of the numbers. So the number of iterations is $O(\lg\lg n)$. The sequential runtime is $T^*(n) = O(n\lg\lg n)$.
Some improvements to be used in an actual sieve implementation.

- Eliminate all even numbers (except 2) from the starting list.
- Eliminate all numbers that are not a multiple of 2 or 3 (except 2 and 3) from the starting list. So the list look like:

  $$2, 3, 5, 7, 11, 13, 17, 19, 23, 25, \ldots$$

  This list can be generated with $3i+2, 3i+4$, where $i = 1, 3, 5, 7, \ldots$.
- We can further eliminate all numbers of the form $30m \pm 5, m \geq 1$, thereby eliminating all spurious multiples of 5.
- Exclusion of multiples of 7 shortens the list by another 14% but no more gains can be made after that since the time to compute the next number in the list exceeds the savings to be had with the list being shorter.

# Sieve of Eratosthenes (contd.)

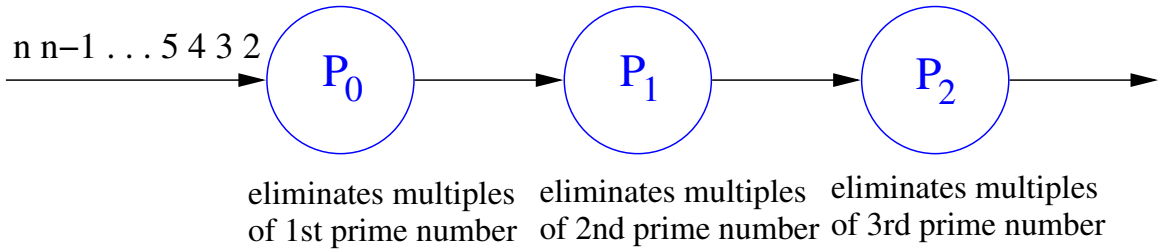
The **Prime Number Theorem** states that we have $\Theta(n/\lg n)$ prime numbers between 1 and $n$. So we will, in general, stop the sieve iterations when the count of the remaining numbers is $\Theta(n/\lg n)$. In each iteration, we are eliminating a constant fraction of the numbers. So the number of iterations is $O(\lg\lg n)$. The sequential runtime is $T^*(n) = O(n\lg\lg n)$.
Some improvements to be used in an actual sieve implementation.

- Eliminate all even numbers (except 2) from the starting list.
- Eliminate all numbers that are not a multiple of 2 or 3 (except 2 and 3) from the starting list. So the list look like:

  $$2, 3, 5, 7, 11, 13, 17, 19, 23, 25, \ldots$$

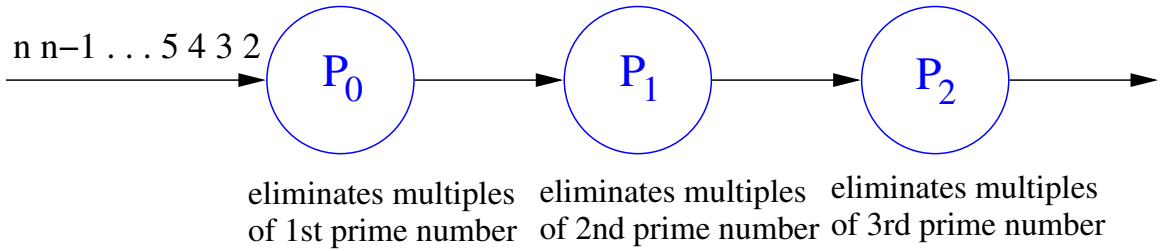
  This list can be generated with $3i+2, 3i+4$, where $i = 1, 3, 5, 7, \ldots$.
- We can further eliminate all numbers of the form $30m \pm 5, m \geq 1$, thereby eliminating all spurious multiples of 5.
- Exclusion of multiples of 7 shortens the list by another 14% but no more gains can be made after that since the time to compute the next number in the list exceeds the savings to be had with the list being shorter.
- We can also use addition/subtractions and avoid multiplications although it it is tricky to set this up properly. For more details, see (*A Practical Sieve Algorithm for Finding Prime Numbers* by Xuedong Luo, Communications of the ACM, Mar 1989, Volume 32, Number 3, pp. 344-346.)

# Pipelined Sieve




A Pipelined Prime Number Sieve

# Pipelined Sieve




A Pipelined Prime Number Sieve

```
sieve(i)
    //Process i, 0 ≤ i ≤ p − 1

    recv(n, P_{i−1})
    recv(prime, P_{i−1})
    for (j=2; j<n; j++) {
        recv(number, P_{i−1})
        if (number == TERMINATOR) break
        if ((number % prime) != 0)
            send(number, P_{i+1})
    }
    send(TERMINATOR, P_{i+1})
```

We need as many processes as the number of primes numbers we will generate. In an implementation we could use a fixed number of processes to simulate the above algorithm.

# Sieve Algorithms for Large Numbers

- Suppose we are dealing with integers with $r$ bits. Then addition/subtraction takes $O(r)$ time and multiplication/division takes $O(r^2)$ time. The sequential runtime becomes $O(r^2 2^r \lg r)$, which is exponential.
- The sieve method is a good way of generating all primes to use in factoring smaller integers but for large integers better methods exist (like the **quadratic sieve**) that takes $O(r^2 2^{r/2})$ time. See *The Art of Computer Programming: Seminumerical Algorithms* (Volume 2) by Don Knuth, Section 4.5.4.

# Example 2: Solving a Triangular System of Linear Equations

$$
\begin{aligned}
a_{n-1,0}x_0 + a_{n-1,1}x_1 + \ldots + a_{n-1,n-1}x_{n-1} &= b_{n-1} \\
\ldots &= \ldots \\
\ldots &= \ldots \\
\ldots &= \ldots \\
a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 &= b_2 \\
a_{1,0}x_0 + a_{1,1}x_1 &= b_1 \\
a_{0,0}x_0 &= b_0
\end{aligned}
$$

The $a$'s and $b$'s are constants and the $x$'s are the unknowns to be found. A simple repeated "back" substitution can be used to solve for $x_0, x_1, \ldots, x_{n-1}$. This is the last step in Gaussian Elimination, a method for solving a general system of linear equations.

# Back Substitution

To solve for the unknowns in a triangular system of equations.

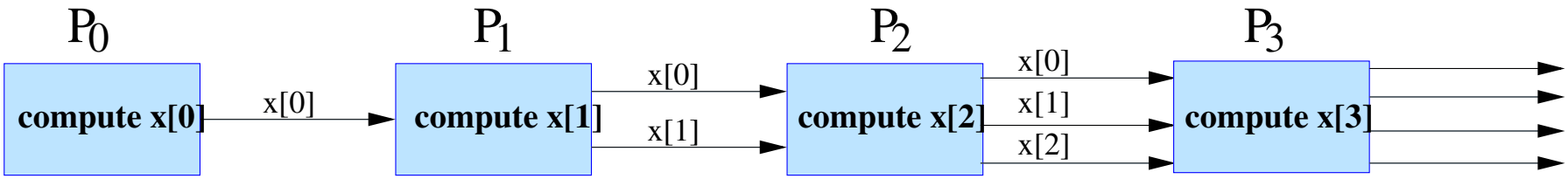
$$
\begin{aligned}
x_0 &= \frac{b_0}{a_{0,0}} \\
x_1 &= \frac{b_1 - a_{1,0}x_0}{a_{1,1}} \\
x_2 &= \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}} \\
&\cdots \\
x_i &= \frac{b_i - \sum_{j=0}^{i-1} a_{i,j}x_j}{a_{i,i}} \\
&\cdots
\end{aligned}
$$

The sequential runtime is $T^*(n) = \Theta(n^2)$.

# Sequential Triangular Solver

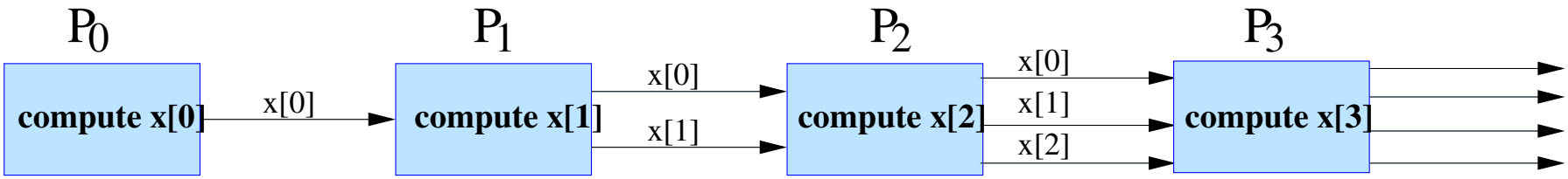
```
triangular_solve(a,b,x)
//a[0..n-1][0..n-1]: matrix of coefficients
//b[0..n-1]: vector of coefficients
//x[0..n-1]: vector of unknowns

    x[0] = b[0]/a[0][0]
    for (i=1; i<n; i++) {
        sum = 0
        for (j=0; j<i; j++) {
            sum = sum + a[i][j] * x[j]
        }
        x[i] = (b[i] - sum)/a[i][i]
    }
```

# Pipelined Triangular Solver




Pipelined Triangular Solver

# Pipelined Triangular Solver




Pipelined Triangular Solver

```
triangular_solve(a,b,x,i)
//i: process id, 0 ≤ i ≤ n − 1
//a[0..n-1][0..n-1]: matrix of coefficients
//b[0..n-1]: vector of coefficients
//x[0..n-1]: vector of unknowns
    sum = 0
    for (j=0; j<i; j++) {
        recv(x[j], P_{i−1})
        send(x[j], P_{i+1})
        sum = sum + a[i][j] * x[j]
    }
    x[i] = (b[i] - sum)/a[i][i]
    send(x[i], P_{i+1})
```

# Analysis of the Pipelined Triangular Solver

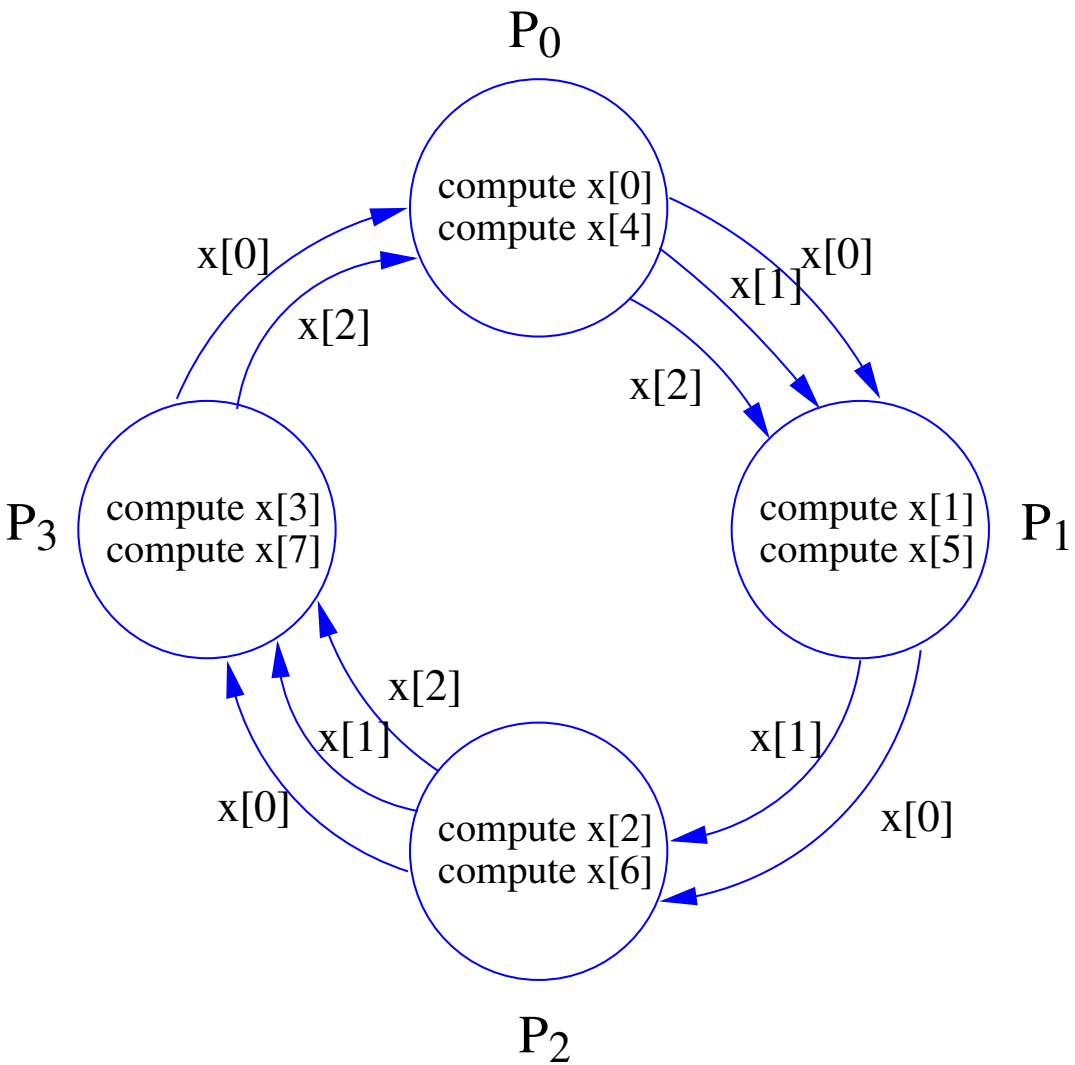
- The $i$th process ($0 < i < n-1$) performs $i$ `recv()`'s, $i$ `send()`'s, $i$ multiply/divide operations, one divide/subtract and a final `send()`– for a total of $2i+1$ communication steps and $2i+2$ computational steps. That gives us a parallel time of $O(n)$, and a good speedup if communication and computation times are balanced.
- Reasonable algorithm for a shared memory machine.
- For a distributed memory machine, we need to adapt the pipelining approach so that it works for a fixed number of processes.

# Parallel Triangular Solver

How would you solve the triangular array of equations on a cluster?

- Adapt pipelining to the cluster.
- Use partitioning.

# Adapting Solutions to a Cluster



# Adapting Solution to a Cluster

- We can also use partitioning for the triangular array of equations. However, if we just assign $n/p$ equations to each process, the work load will not be balanced. The following table shows an example of balancing the load by assigning different number of equations to different processes.

| eqn. no. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| work load | 1 | 3 | 6 | 10 | 15 | 21 | 28 | 36 | 45 | 55 | 66 | 78 | 81 | 95 | 110 | 136 |
| assignment | Process 0 | | | | | | | | | Process 1 | | | | Process 2 | | |

# Adapting Solution to a Cluster

- We can also use partitioning for the triangular array of equations. However, if we just assign $n/p$ equations to each process, the work load will not be balanced. The following table shows an example of balancing the load by assigning different number of equations to different processes.

| eqn. no. | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| work load | 1 | 3 | 6 | 10 | 15 | 21 | 28 | 36 | 45 | 55 | 66 | 78 | 81 | 95 | 110 | 136 |
| assignment | Process 0 | | | | | | | | | Process 1 | | | | Process 2 | | |

- The work load for the $i$th equation is $O(i)$ steps. The second line shows the cumulative work load for processing the first $i$ equations. Since we have 3 processes, we divide the total workload by three, which is 45. Thus process 0 handles equations 1 through 9, process 1 handles equations 10 through 13 and process 2 the rest to balance the work load.