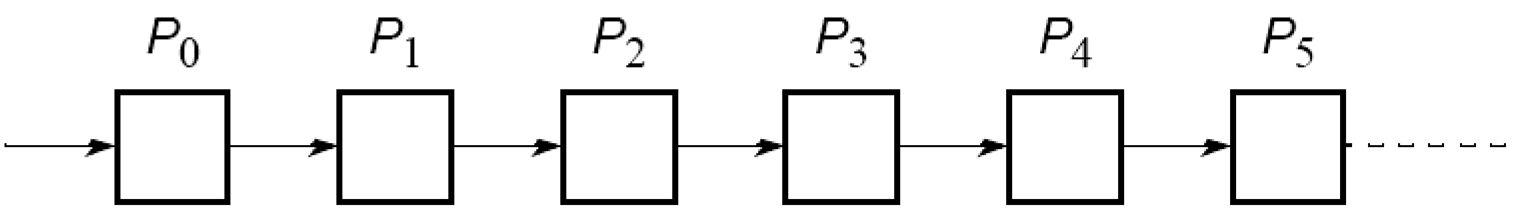
Pipelined Computations

Pipelined Computations

Problem divided into a series of tasks that have to be completed one after the other (the basis of sequential programming). Each task executed by a separate process or processor.



Example

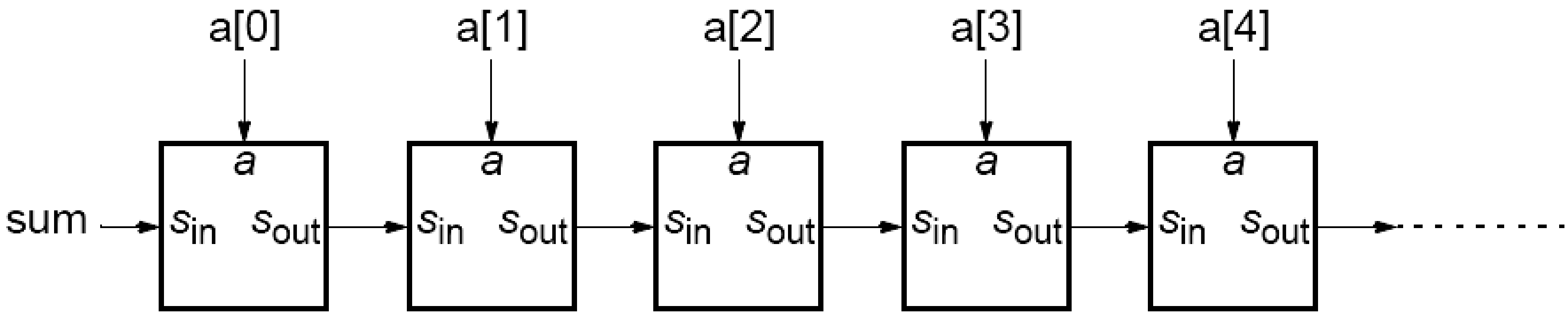
Add all the elements of array **a** to an accumulating sum:

```
for (i = 0; i < n; i++)
sum = sum + a[i];
```

The loop could be "unfolded" to yield

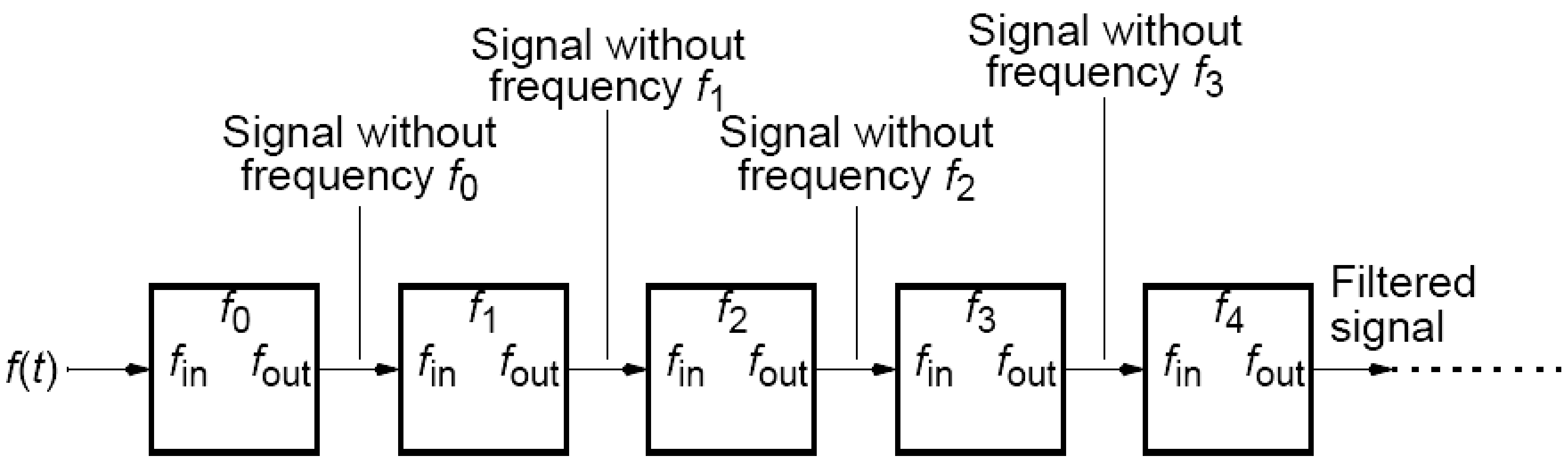
```
sum = sum + a[0];
sum = sum + a[1];
sum = sum + a[2];
sum = sum + a[3];
sum = sum + a[4];
          .
          .
          .
```

Pipeline for an unfolded loop



Another Example

Frequency filter - Objective to remove specific frequencies (f_0, f_1, f_2,f_3, etc.) from a digitized signal, $f(t)$.
Signal enters pipeline from left:



Where pipelining can be used to good effect

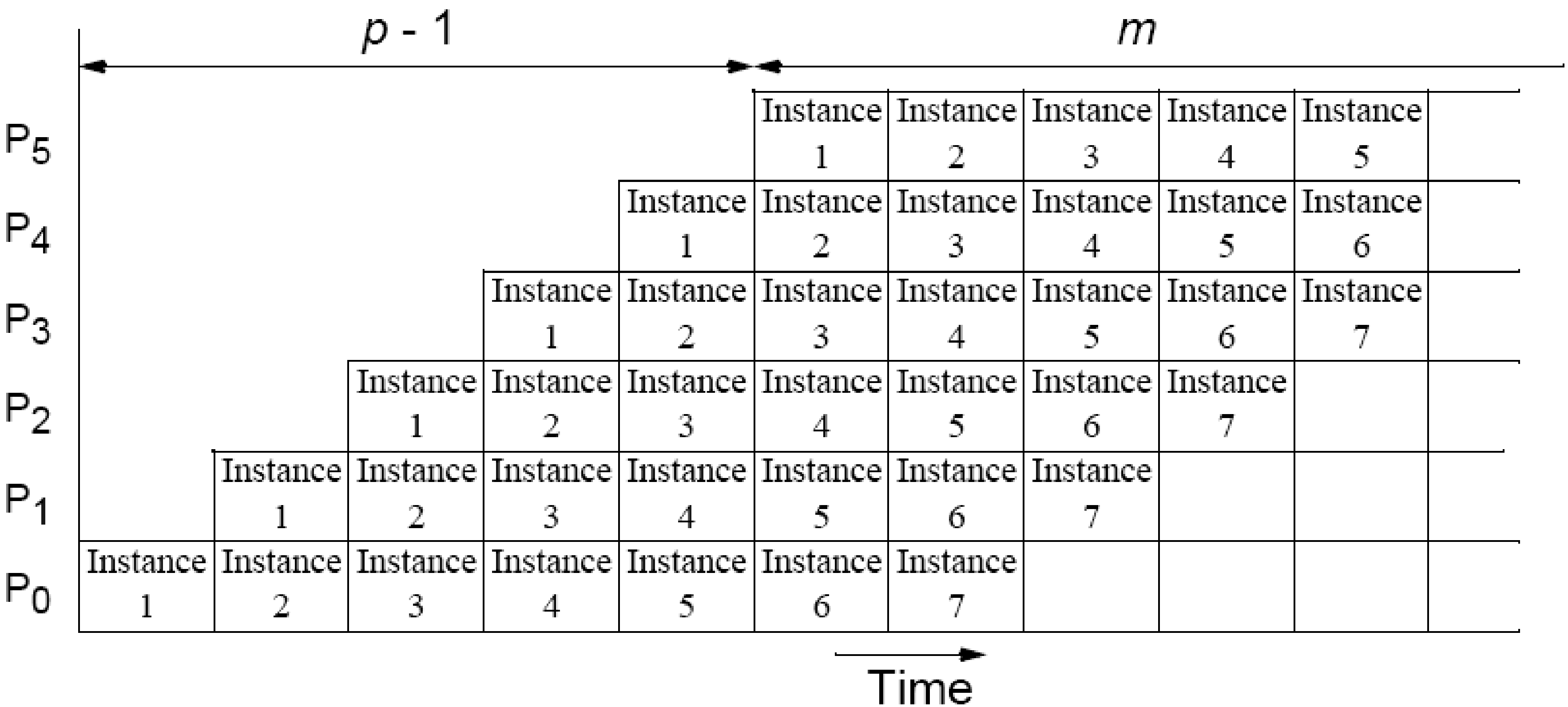
Assuming problem can be divided into a series of sequential tasks, pipelined approach can provide increased execution speed under the following three types of computations:

1. If more than one instance of the complete problem is to be Executed

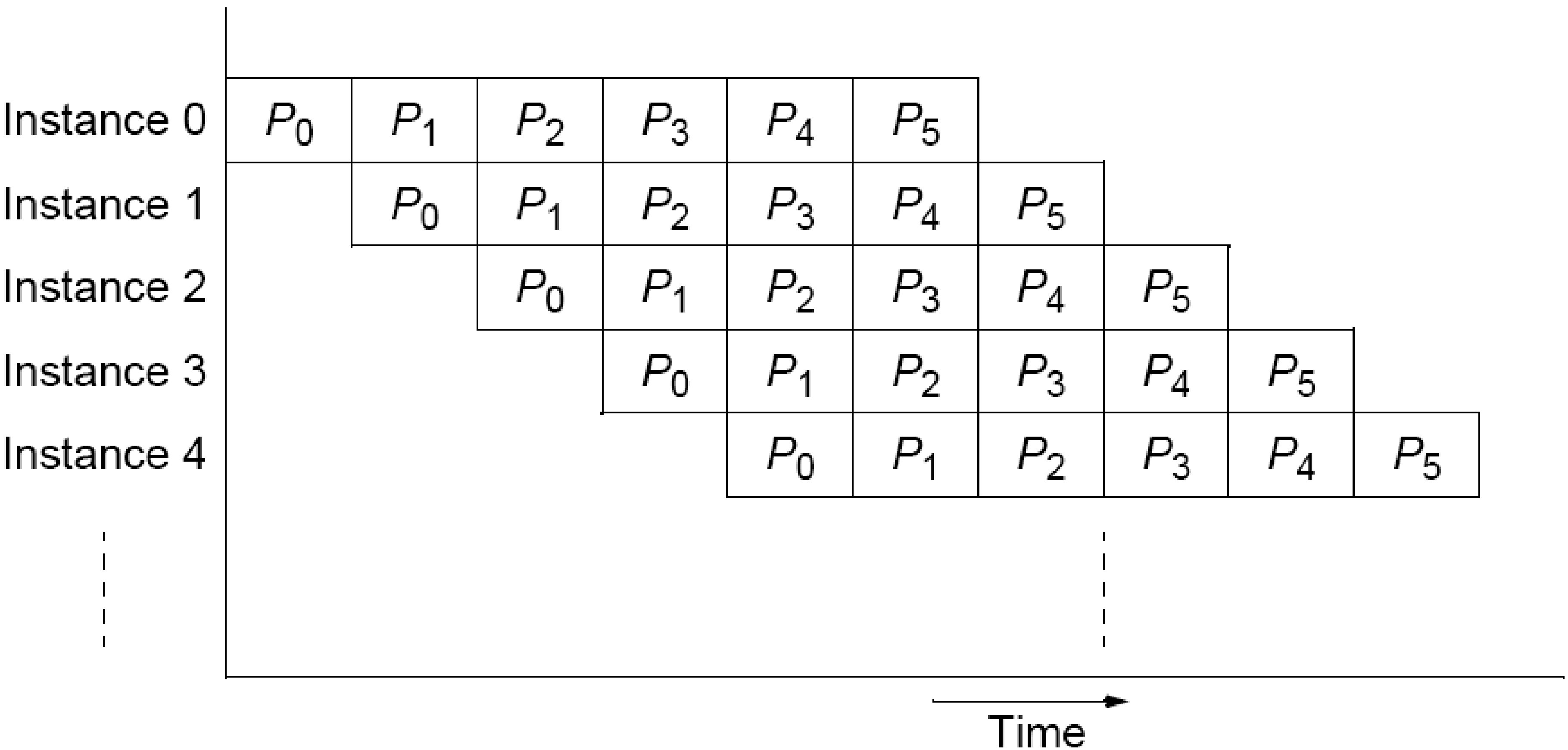
2. If a series of data items must be processed, each requiring multiple operations

3. If information to start next process can be passed forward before process has completed all its internal operations

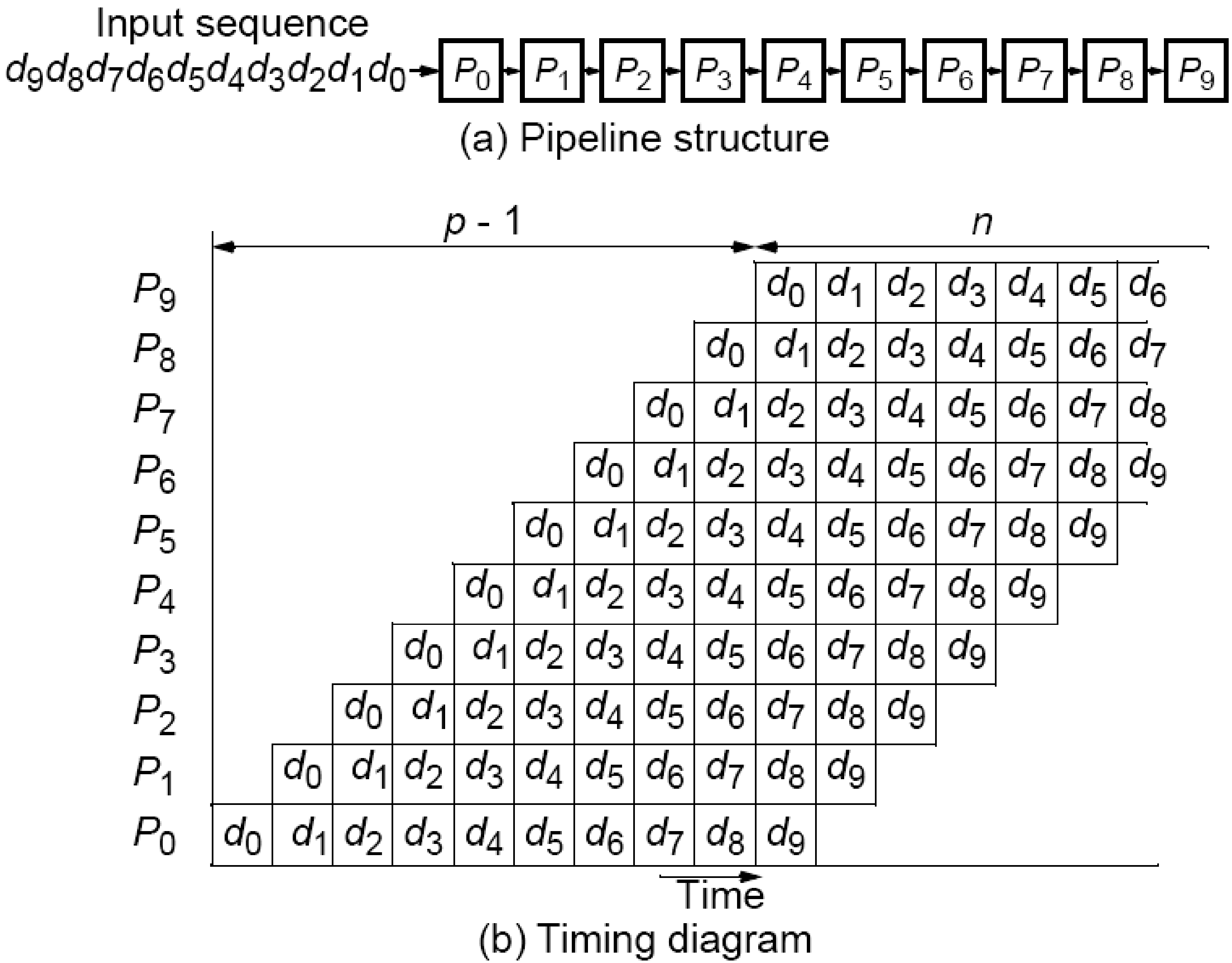
“Type 1” Pipeline Space-Time Diagram



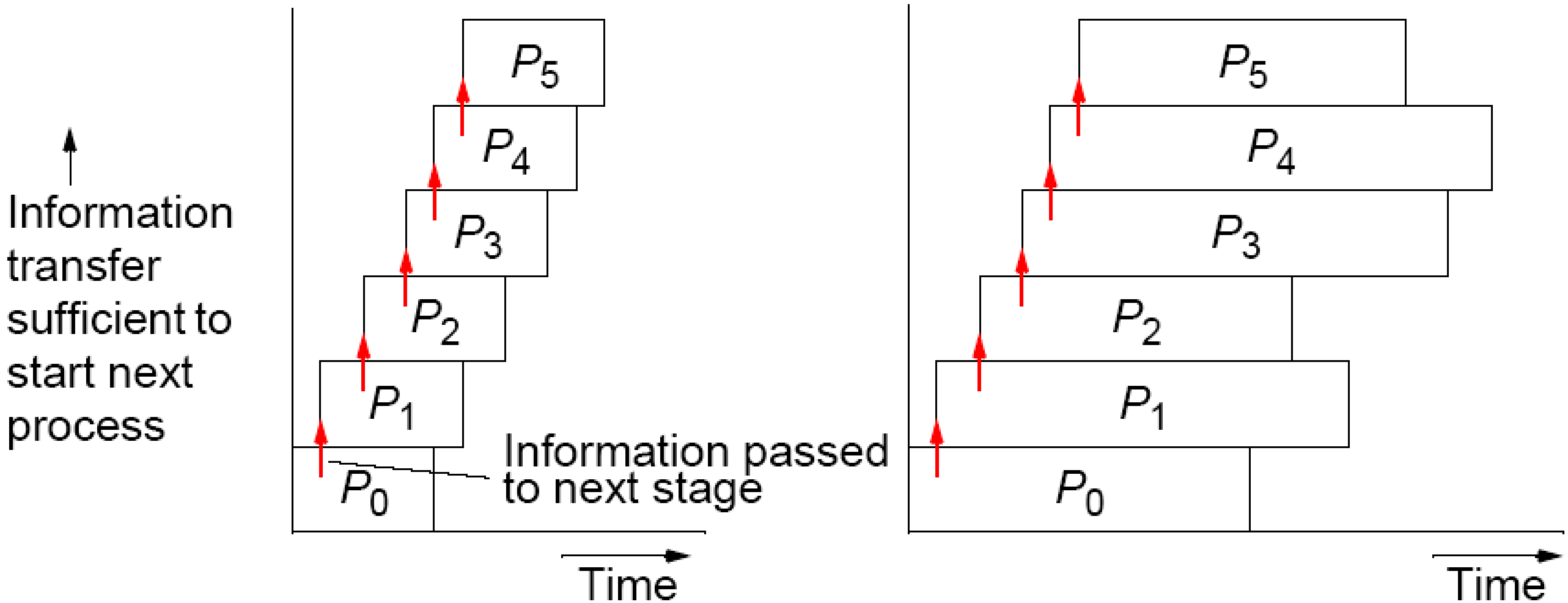
Alternative space-time diagram



“Type 2” Pipeline Space-Time Diagram



“Type 3” Pipeline Space-Time Diagram

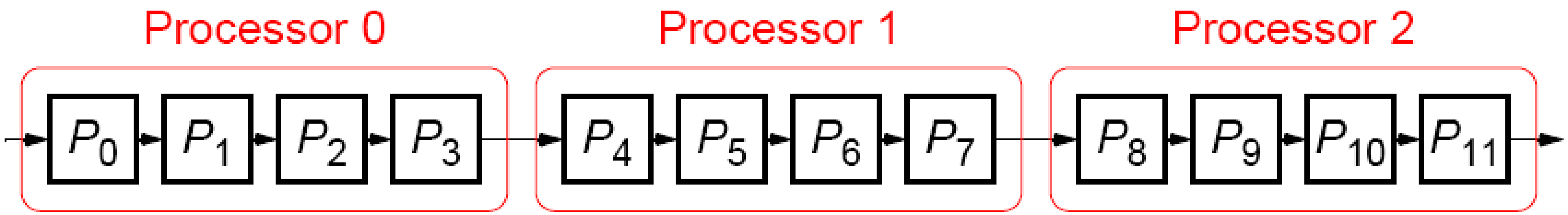


(a) Processes with the same execution time

(b) Processes not with the same execution time

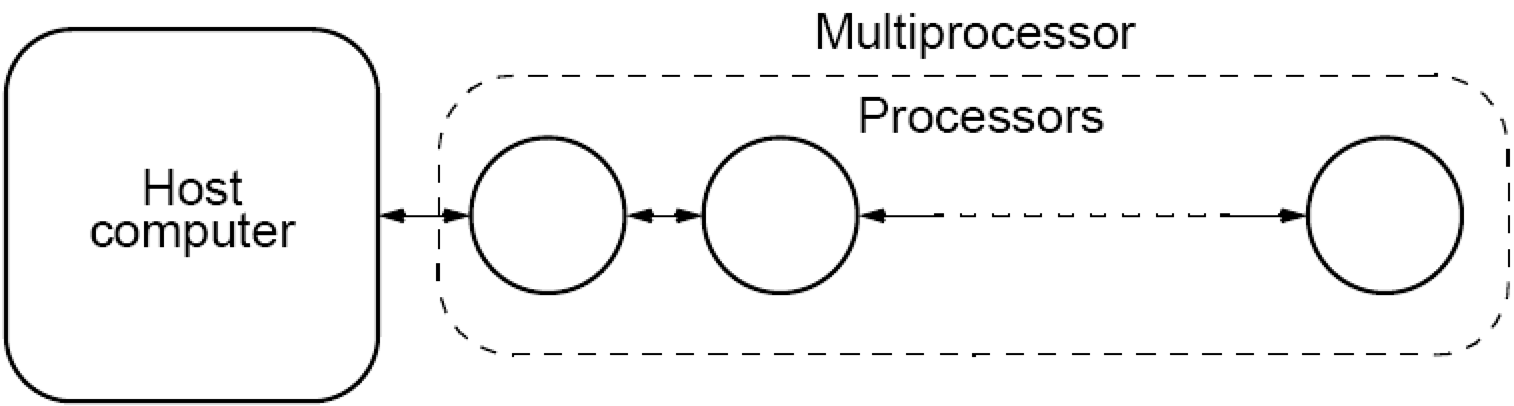
Pipeline processing where information passes to next stage before previous state completed.

If the number of stages is larger than the number of processors in any pipeline, a group of stages can be assigned to each processor:



Computing Platform for Pipelined Applications

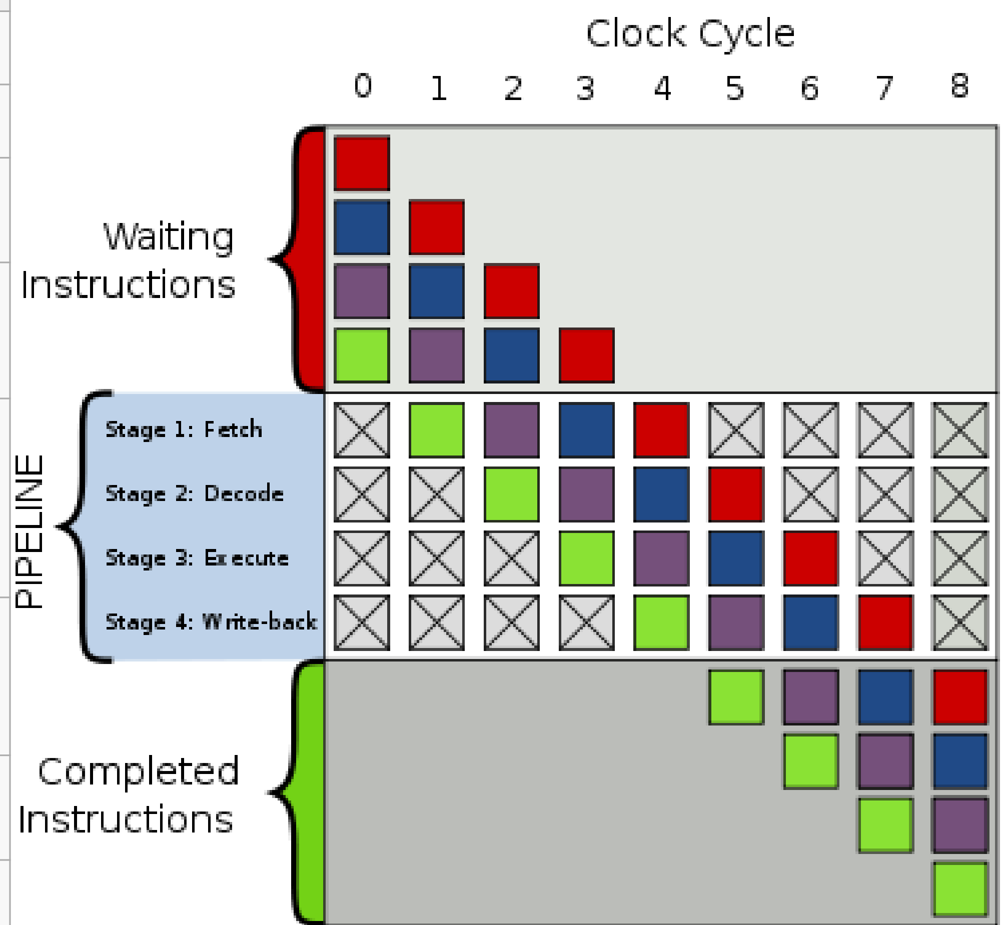
Multiprocessor system with a line configuration



Strictly speaking pipeline may not be the best structure for a cluster - however a cluster with switched direct connections, as most have, can support simultaneous message passing.

Pipelined Instructions on a Processor

Time	Execution
0	Four instructions are waiting to be executed
1	•The green instruction is fetched from memory
2	•The green instruction is decoded •The purple instruction is fetched from memory
3	•The green instruction is executed (actual operation is performed) •The purple instruction is decoded •The blue instruction is fetched
4	•The green instruction's results are written back to the register file or memory •The purple instruction is executed •The blue instruction is decoded •The red instruction is fetched
5	•The green instruction is completed •The purple instruction is written back •The blue instruction is executed •The red instruction is decoded
6	•The purple instruction is completed •The blue instruction is written back •The red instruction is executed
7	•The blue instruction is completed •The red instruction is written back
8	•The red instruction is completed
9	All four instructions are executed



Source Wikipedia

Intel Sandybridge has a 14 to 19 stage instruction pipeline

Example possible stages for Multiply

- Real numbers can be represented as mantissa and exponent in a "normalized" representation, e.g.: $s*0.m * 10^{e}$ with

  Sign **s={-1,1}**

  Mantissa **m** which does not contain 0 in leading digit

  Exponent **e** some positive or negative integer

- Multiply two real numbers **r1*r2 = r3**
- **$r1=s1*0.m1 * 10^{e1}$, $r2=s2*0.m2 * 10^{e2}$:**

  $s1*0.m1 * 10^{e1} * s2*0.m2 * 10^{e2}$

→ **$(s1*s2) * (0.m1*0.m2) * 10^{(e1+e2)}$**

→ Normalize result: **$s3* 0.m3 * 10^{e3}$**

Source - D. Fey and G. Wellein

5-stage Multiplication-Pipeline: A(i)=B(i)*C(i) ; i=1,...,N

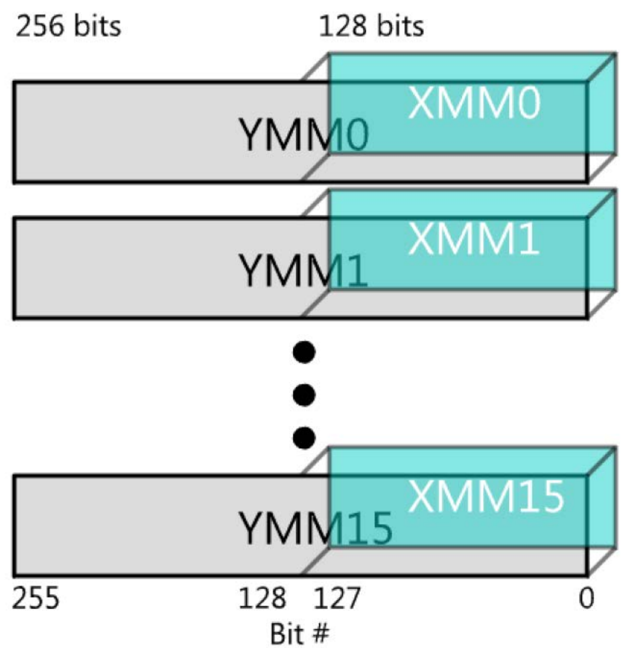
Stage	Cycle: 1	2	3	4	5	6	...	N+4
Separate Mant. / Exp.	B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	B(5) C(5)	B(6) C(6)	...	
Mult. Mantissa		B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	B(5) C(5)	...	
Add. Exponents			B(1) C(1)	B(2) C(2)	B(3) C(3)	B(4) C(4)	...	
Normal. Result				**A(1)**	**A(2)**	B(3) C(3)	...	
Insert Sign					**A(1)**	**A(2)**	...	**A(N)**

First result is available after 5 cycles (=latency of pipeline)!
After that one instruction is completed in each cycle

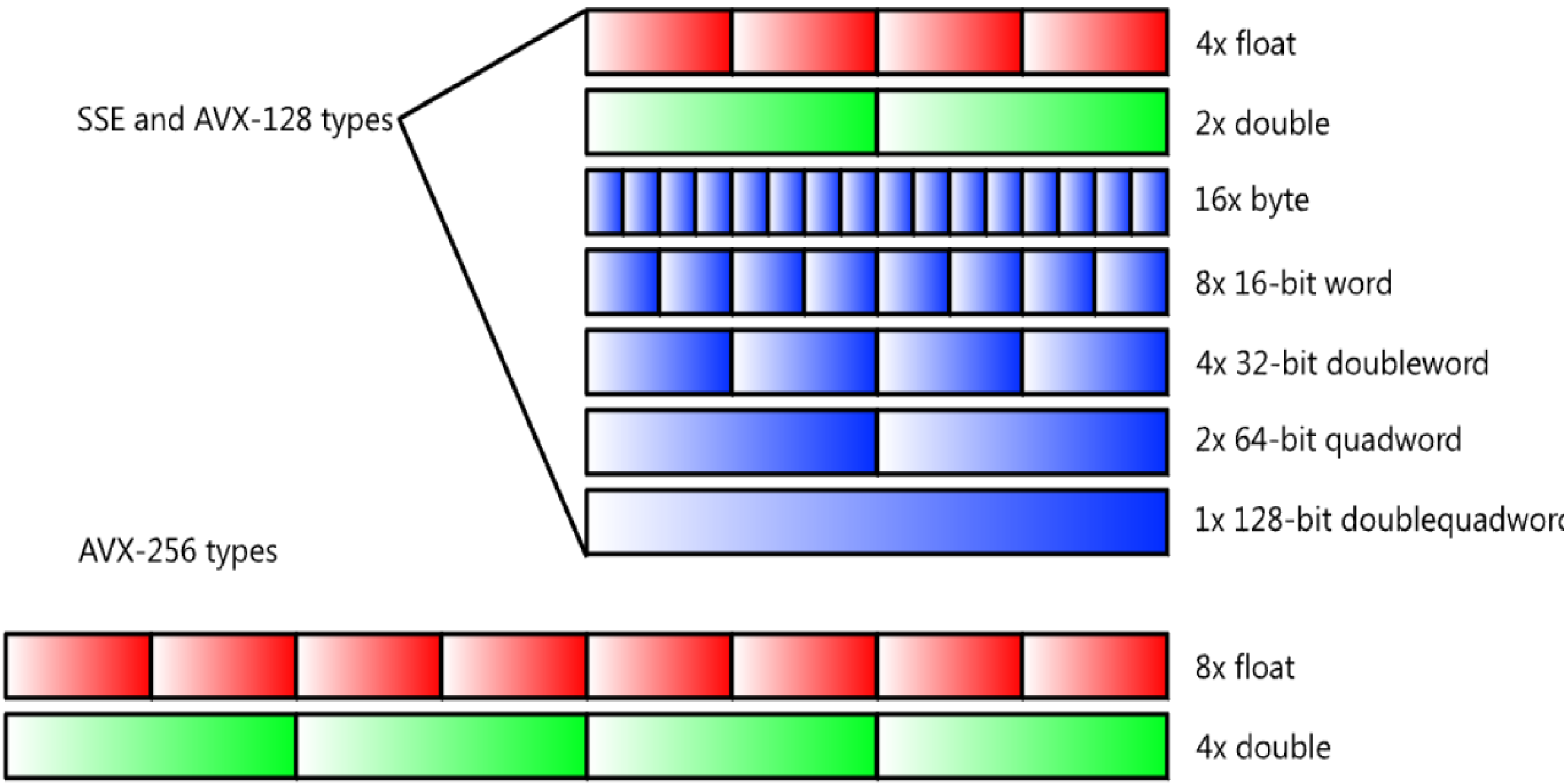
Source D. Fey and G. Wellein

Pipelining and SIMD Calculations in Modern CPUs Intel Advanced Vector Extensions AVX

- 128 bit Instructions previously used SSE expanded to 256 bit
- Three and four operands available.
- Faster operations A = A*B+C (Fused multiply add) and new A = B*C
- New instruction set (Vex)
- Builds on earlier SSE
- Extensions to 512 bit coming
- Can only get close to peak performance if AVX used
- 16 256 bit registers YMM aliased over old XMM SSE registers
- Four floating point operations concurrently in a pipeline



Intel AVX vs SSE



Intel AVX SIMD Mode

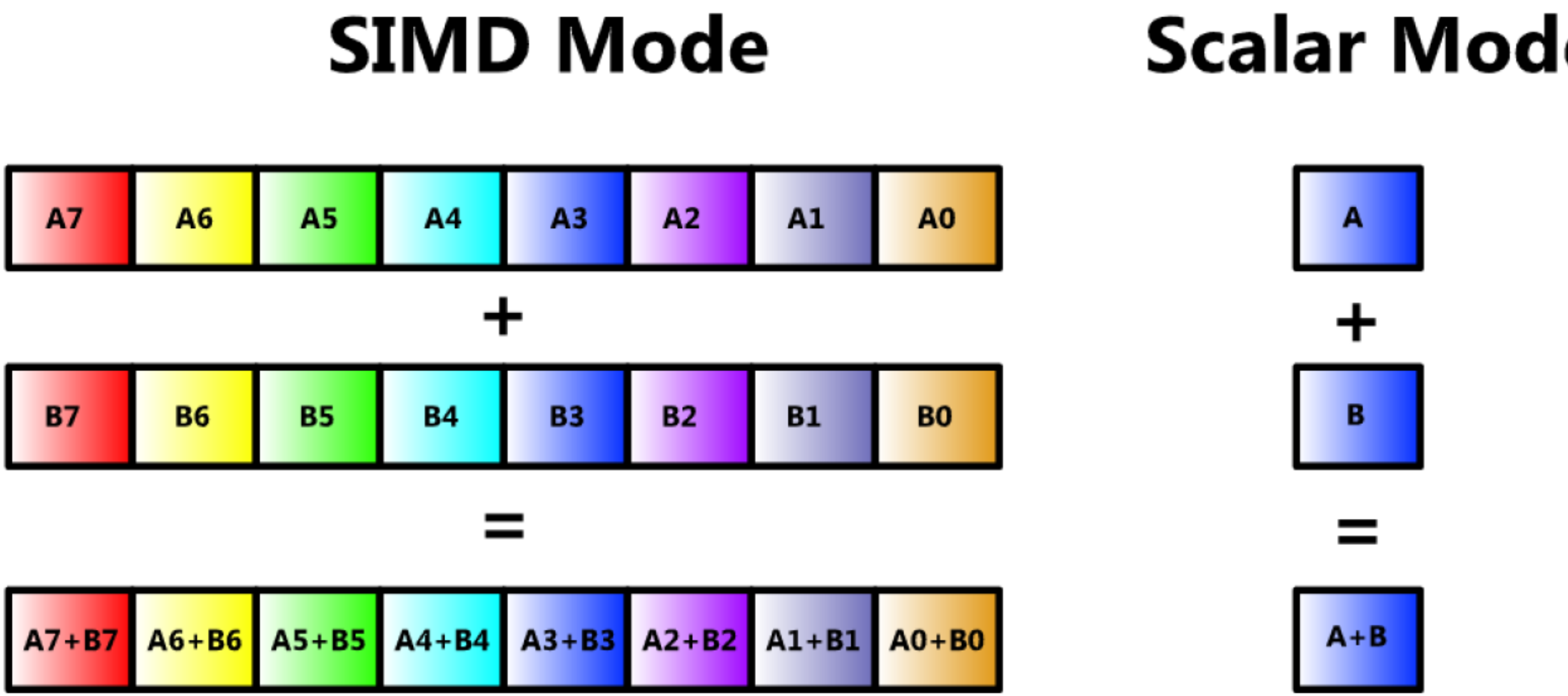
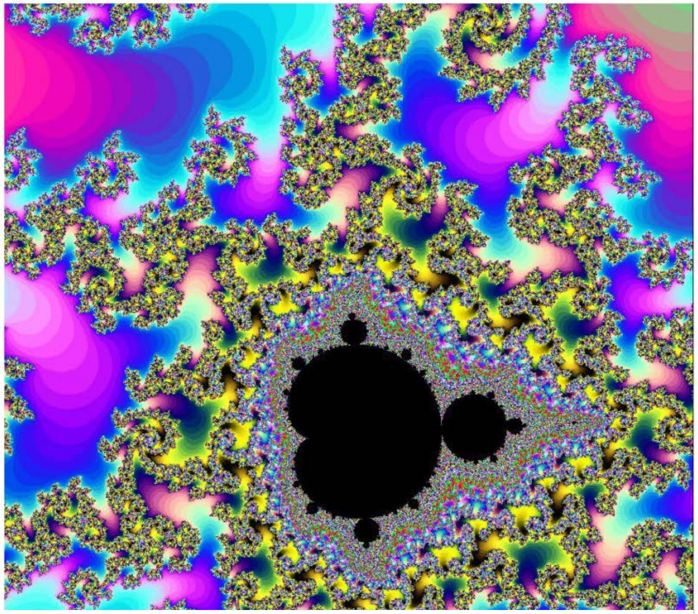


Figure 3. SIMD versus scalar operations

Mandelbrot Set Example

- Standard Code



```
// simple code to compute Mandelbrot in C++
#include <complex>
void MandelbrotCPU(float x1, float y1, float x2, float y2,
                   int width, int height, int maxIters, unsigned short * image)
{
        float dx = (x2-x1)/width, dy = (y2-y1)/height;
        for (int j = 0; j < height; ++j)
                for (int i = 0; i < width; ++i)
                {
                        complex<float> c (x1+dx*i, y1+dy*j), z(0,0);
                        int count = -1;
                        while ((++count < maxIters) && (norm(z) < 4.0))
                                z = z*z+c;
                        *image++ = count;
                }
}
```

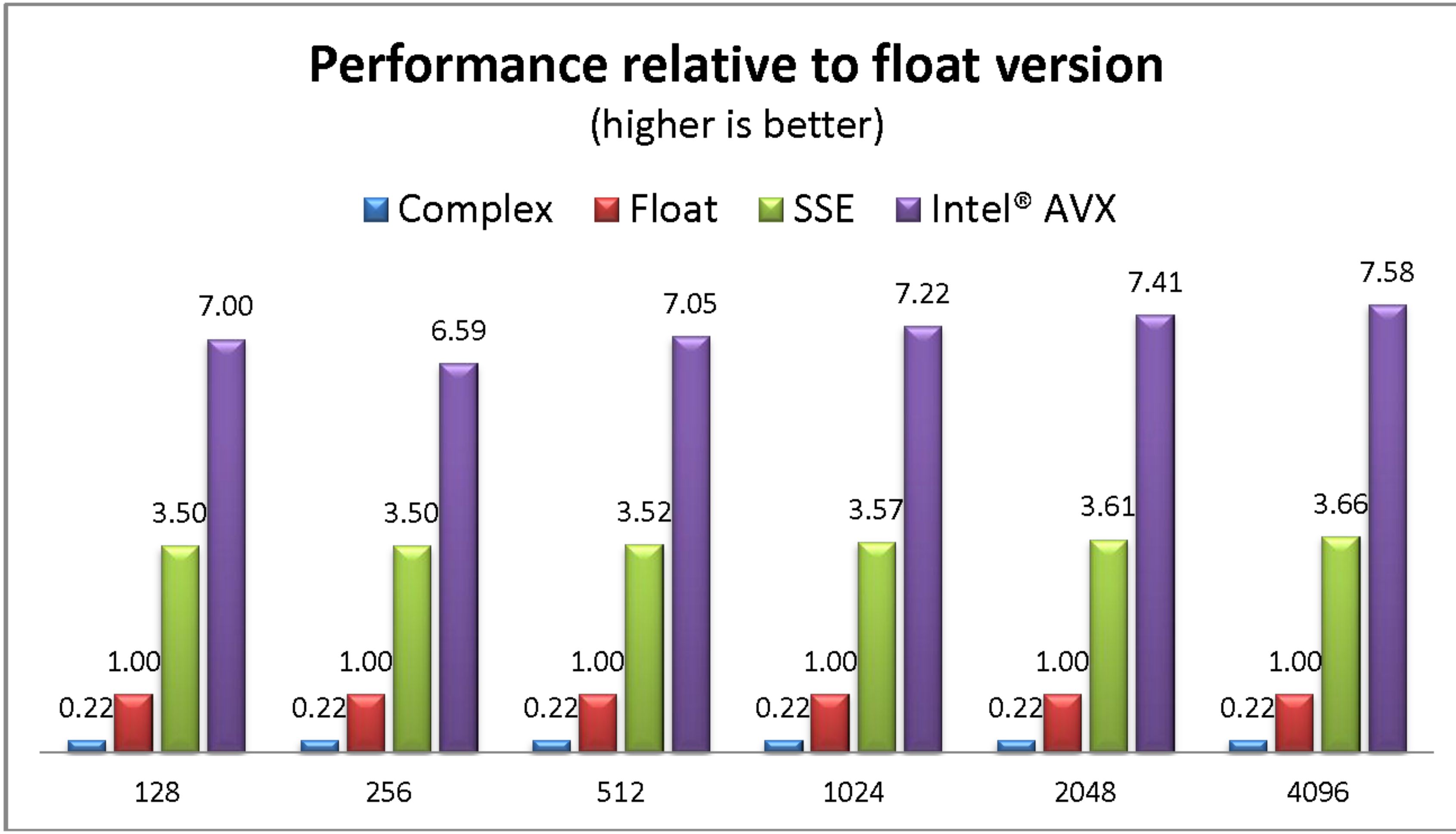
AVX Mandelbrot Code Using AVX Instructions

Listing 5. Intel® AVX-intrinsic Mandelbrot Implementation

```
float dx = (x2-x1)/width;
float dy = (y2-y1)/height;
// round up width to next multiple of 8
int roundedWidth = (width+7) & ~7UL;

float constants[] = {dx, dy, x1, y1, 1.0f, 4.0f};
__m256 ymm0 = _mm256_broadcast_ss(constants);   // all dx
__m256 ymm1 = _mm256_broadcast_ss(constants+1); // all dy
__m256 ymm2 = _mm256_broadcast_ss(constants+2); // all x1
__m256 ymm3 = _mm256_broadcast_ss(constants+3); // all y1
__m256 ymm4 = _mm256_broadcast_ss(constants+4); // all 1's (iter increments)
__m256 ymm5 = _mm256_broadcast_ss(constants+5); // all 4's (comparisons)

float incr[8]={0.0f,1.0f,2.0f,3.0f,4.0f,5.0f,6.0f,7.0f}; // used to reset the i position when
 j increases
__m256 ymm6 = _mm256_xor_ps(ymm0,ymm0); // zero out j counter (ymm0 is just a dummy)

for (int j = 0; j < height; j+=1)
{
       __m256 ymm7  = _mm256_load_ps(incr);  // i counter set to 0,1,2,..,7
       for (int i = 0; i < roundedWidth; i+=8)
       {
```

```
        __m256 ymm8 = _mm256_mul_ps(ymm7, ymm0);  // x0 = (i+k)*dx
        ymm8 = _mm256_add_ps(ymm8, ymm2);         // x0 = x1+(i+k)*dx
        __m256 ymm9 = _mm256_mul_ps(ymm6, ymm1);  // y0 = j*dy
        ymm9 = _mm256_add_ps(ymm9, ymm3);         // y0 = y1+j*dy
        __m256 ymm10 = _mm256_xor_ps(ymm0,ymm0);  // zero out iteration counter
        __m256 ymm11 = ymm10, ymm12 = ymm10;        // set initial xi=0, yi=0

        unsigned int test = 0;
        int iter = 0;
        do
        {
                __m256 ymm13 = _mm256_mul_ps(ymm11,ymm11); // xi*xi
                __m256 ymm14 = _mm256_mul_ps(ymm12,ymm12); // yi*yi
                __m256 ymm15 = _mm256_add_ps(ymm13,ymm14); // xi*xi+yi*yi

                // xi*xi+yi*yi < 4 in each slot
                ymm15 = _mm256_cmp_ps(ymm15,ymm5, _CMP_LT_OQ);
                // now ymm15 has all 1s in the non overflowed locations

    test = _mm256_movemask_ps(ymm15)&255;       // lower 8 bits are comparisons
                ymm15 = _mm256_and_ps(ymm15,ymm4);
                // get 1.0f or 0.0f in each field as counters
                // counters for each pixel iteration
                ymm10 = _mm256_add_ps(ymm10,ymm15);

                ymm15 = _mm256_mul_ps(ymm11,ymm12);        // xi*yi
                ymm11 = _mm256_sub_ps(ymm13,ymm14);        // xi*xi-yi*yi
                ymm11 = _mm256_add_ps(ymm11,ymm8);         // xi <- xi*xi-yi*yi+x0 done!
                ymm12 = _mm256_add_ps(ymm15,ymm15);        // 2*xi*yi
                ymm12 = _mm256_add_ps(ymm12,ymm9);         // yi <- 2*xi*yi+y0

                ++iter;
        } while ((test != 0) && (iter < maxIters));

        // convert iterations to output values
        __m256i ymm10i = _mm256_cvtps_epi32(ymm10);

        // write only where needed
        int top = (i+7) < width? 8: width&7;
        for (int k = 0; k < top; ++k)
                image[i+k+j*width] = ymm10i.m256i_i16[2*k];

        // next i position - increment each slot by 8
        ymm7 = _mm256_add_ps(ymm7, ymm5);
        ymm7 = _mm256_add_ps(ymm7, ymm5);
    }
    ymm6 = _mm256_add_ps(ymm6,ymm4); // increment j counter
}
```

Mandelbrot Set Performance

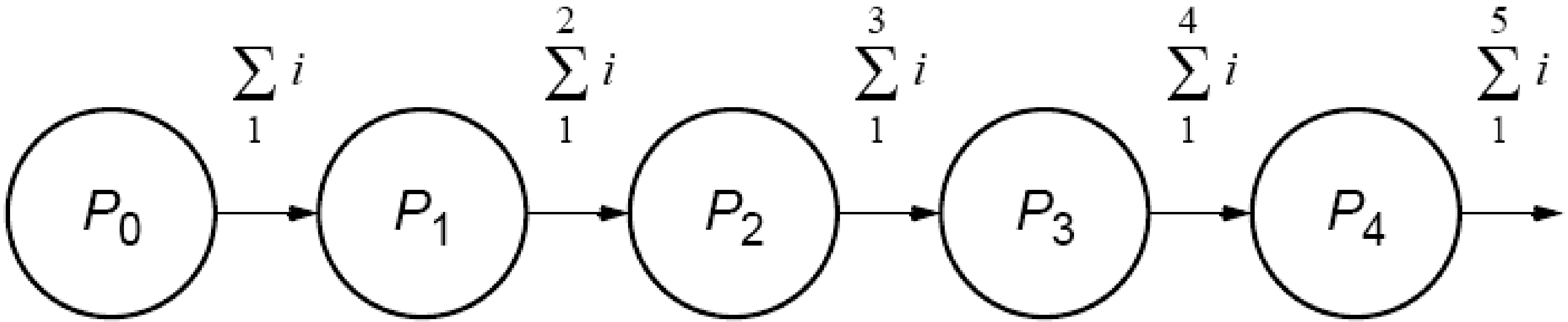


For more information see presentation by Gropp et al.

Example Pipelined Solutions
(Examples of each type of computation)

Pipeline Program Examples

Adding Numbers



Type 1 pipeline computation

Basic code for process *Pi* :

```
recv(&accumulation, Pi-1);
accumulation = accumulation + number;
send(&accumulation, Pi+1);
```

except for the first process, *P*0, which is

```
send(&number, P1);
```

and the last process, *Pn*-1, which is

```
recv(&number, Pn-2);
accumulation = accumulation + number;
```


SPMD program

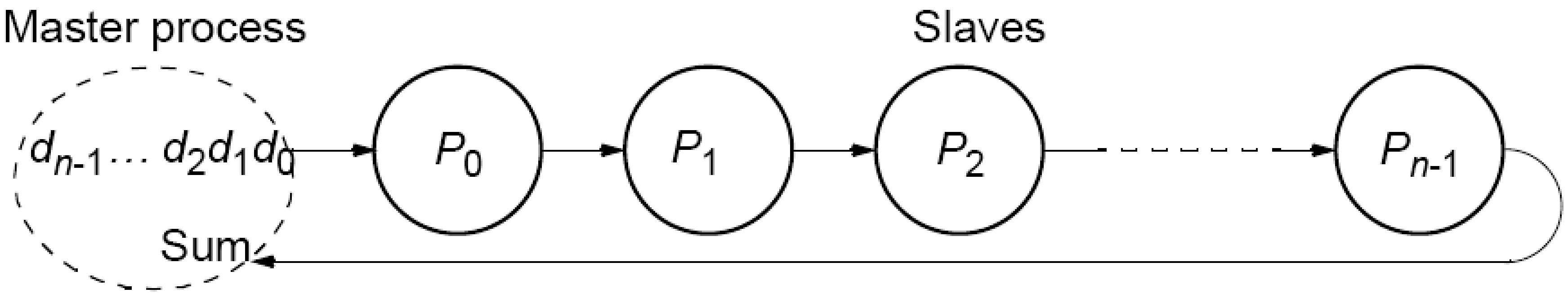
```
if (process > 0) {
        recv(&accumulation, Pi-1);
        accumulation = accumulation + number;
}
if (process < n-1)
        send(&accumulation, P i+1);
```

The final result is in the last process.

Instead of addition, other arithmetic operations could be done.

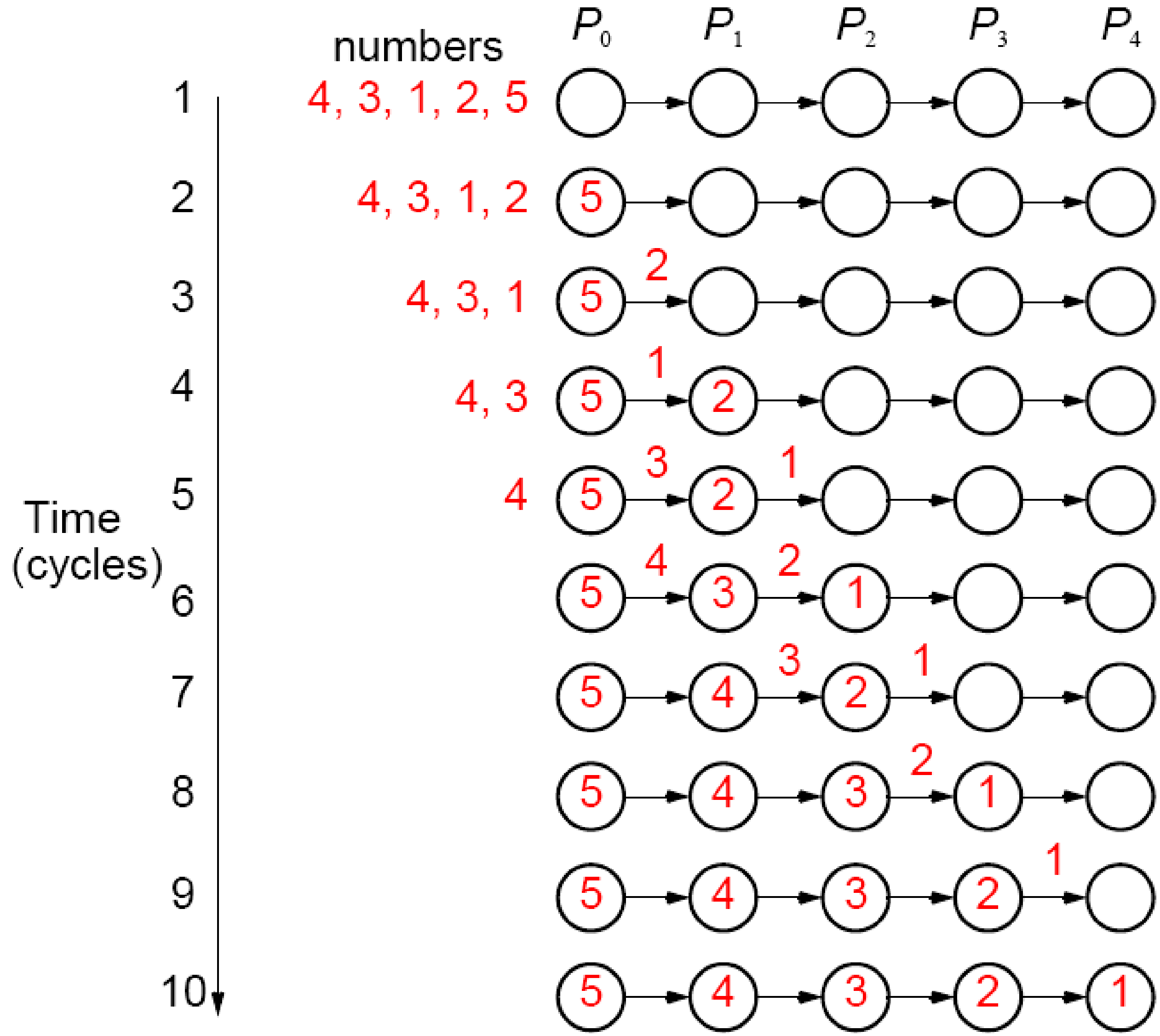
Pipelined addition numbers

Master process and ring configuration

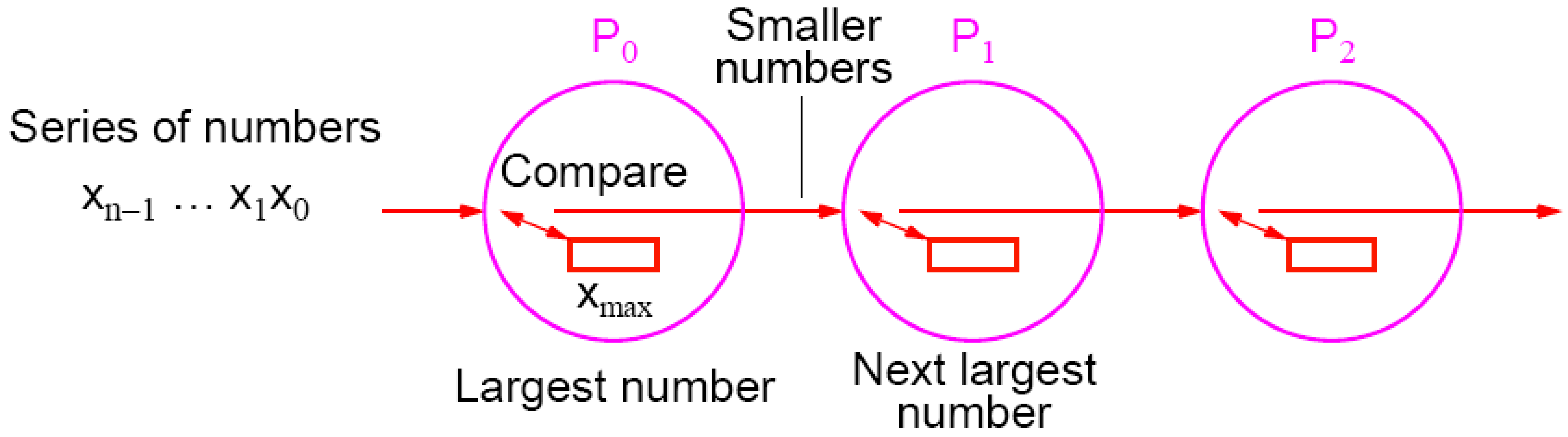


Sorting Numbers

A parallel version of *insertion sort*.



Pipeline for sorting using insertion sort



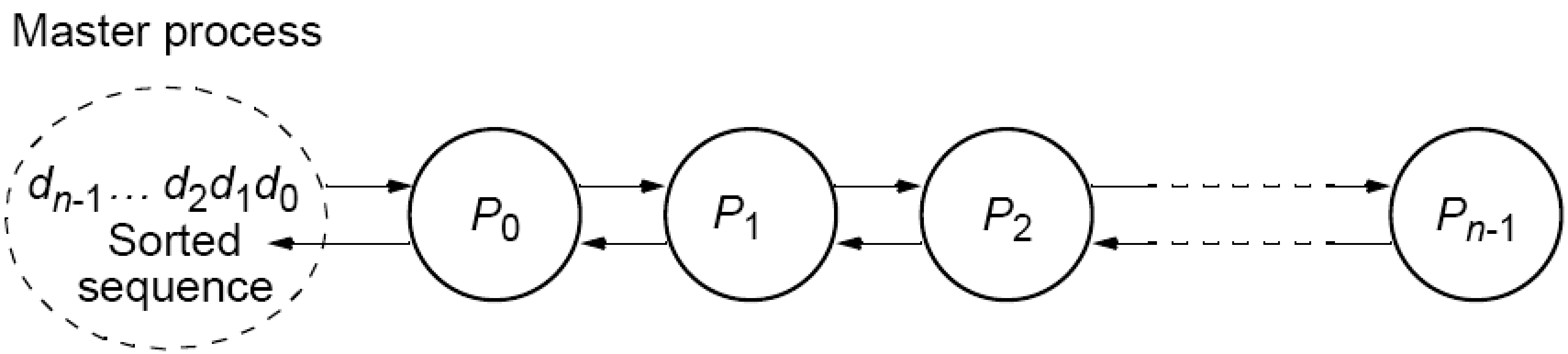
Type 2 pipeline computation

The basic algorithm for process *Pi* is

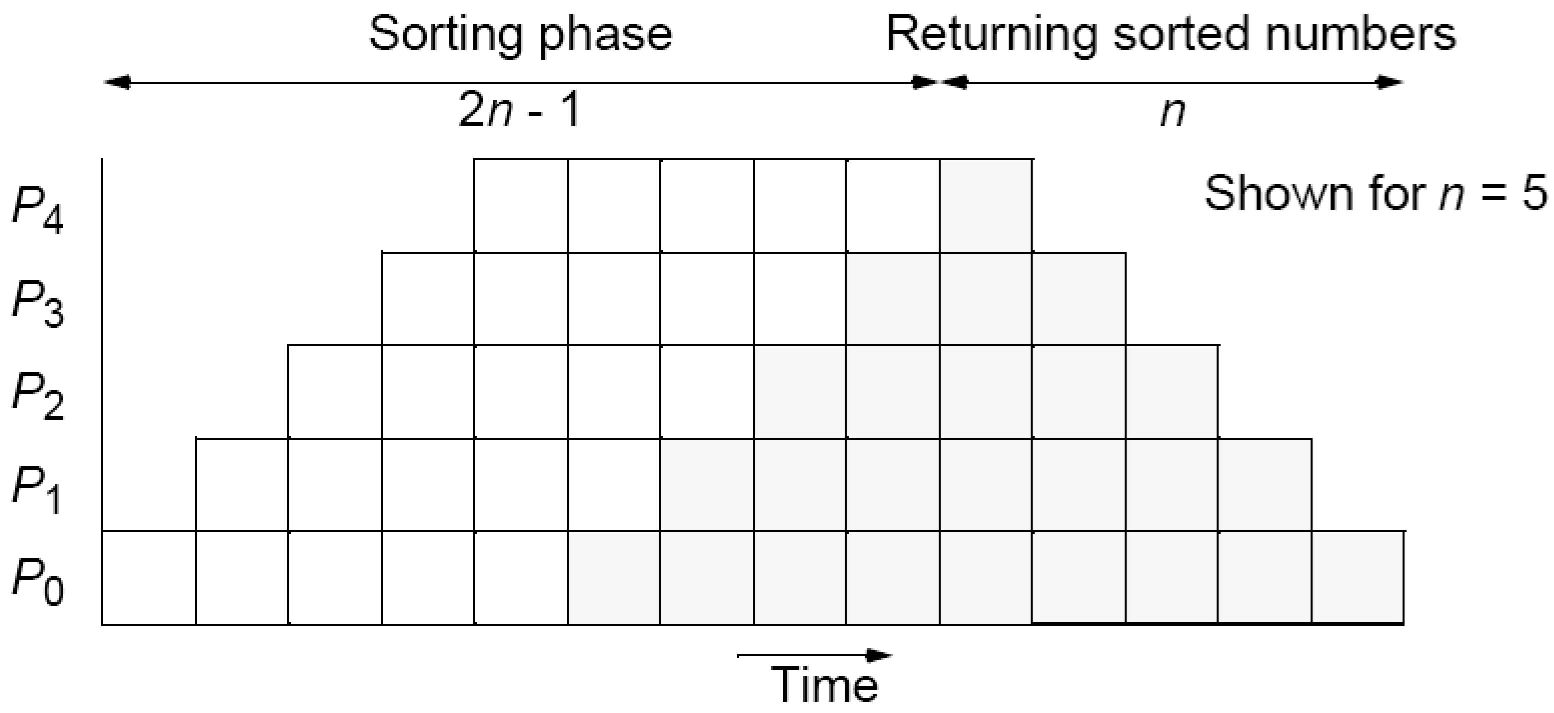
```
recv(&number, Pi-1);
if (number > x) {
        send(&x, Pi+1);
        x = number;
} else send(&number, Pi+1);
```

With n numbers, number ith process is to accept = n - i.
Number of passes onward = n - i - 1
Hence, a simple loop could be used.

Insertion sort with results returned to master process using bidirectional line configuration

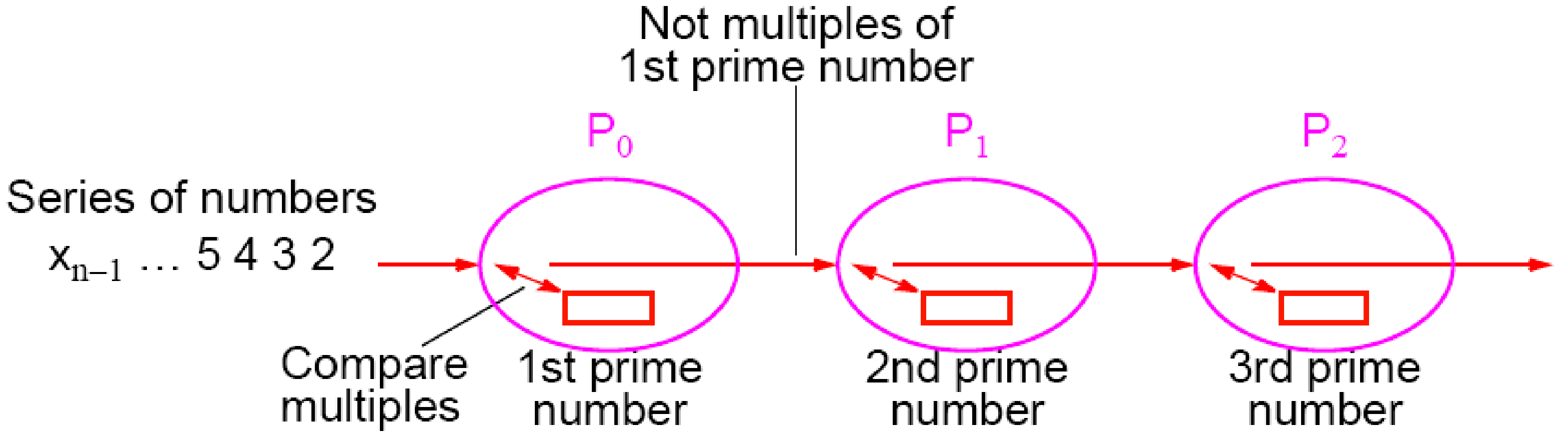


Insertion sort with results returned



Prime Number Generation
Sieve of Eratosthenes

- Series of all integers generated from 2.
- First number, 2, is prime and kept.
- All multiples of this number deleted as they cannot be prime.
- Process repeated with each remaining number.
- The algorithm removes non-primes, leaving only primes.



Type 2 pipeline computation

The code for a process, *Pi*, could be based upon

```
recv(&x, Pi-1);
/* repeat following for each number */
recv(&number, Pi-1);
if ((number % x) != 0) send(&number, P i+1);
```

Each process will not receive the same number of numbers and is not known beforehand. Use a “terminator” message, which is sent at the end of the sequence:

```
recv(&x, Pi-1);
for (i = 0; i < n; i++) {
      recv(&number, Pi-1);
      If (number == terminator) break;
       (number % x) != 0) send(&number, P i+1);
}
```

Solving a System of Linear Equations

Upper-triangular form

$$a_{n-1,0}x_0 + a_{n-1,1}x_1 + a_{n-1,2}x_2 \quad \ldots \quad + a_{n-1,n-1}x_{n-1} = b_{n-1}$$

$$\vdots$$

$$a_{2,0}x_0 + a_{2,1}x_1 + a_{2,2}x_2 = b_2$$

$$a_{1,0}x_0 + a_{1,1}x_1 = b_1$$

$$a_{0,0}x_0 = b_0$$

where *a’s* and *b’s* are constants and *x*’s are unknowns to be found.

Back Substitution

First, unknown x_0 is found from last equation; i.e.,

$$x_0 = \frac{b_0}{a_{0,0}}$$

Value obtained for x_0 substituted into next equation to obtain x_1; i.e.,

$$x_1 = \frac{b_1 - a_{1,0}x_0}{a_{1,1}}$$

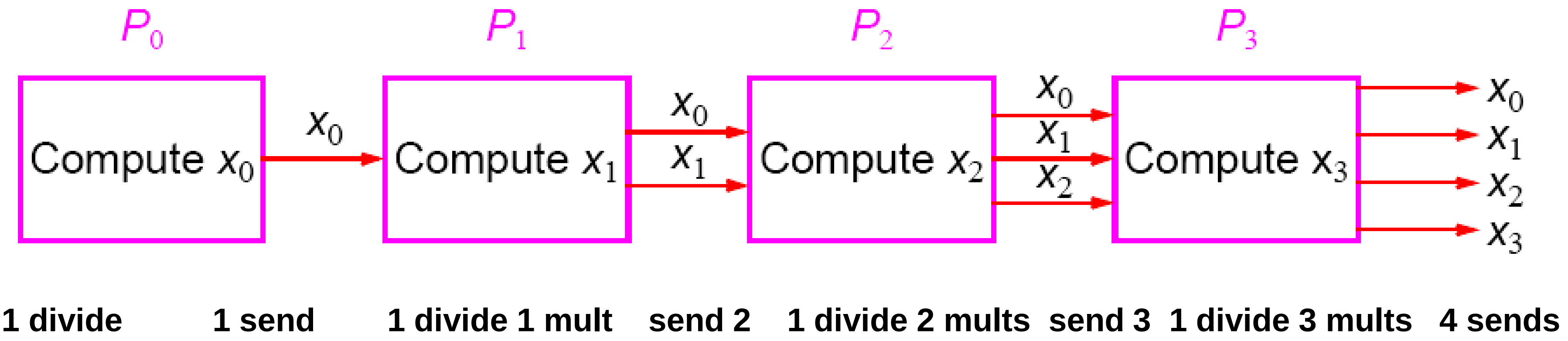
Values obtained for x_1 and x_0 substituted into next equation to obtain x_2:

$$x_2 = \frac{b_2 - a_{2,0}x_0 - a_{2,1}x_1}{a_{2,2}}$$

and so on until all the unknowns are found.

Pipeline Solution

First pipeline stage computes x_0 and passes x_0 onto the second stage, which computes x_1 from x_0 and passes both x_0 and x_1 onto the next stage, which computes x_2 from x_0 and x_1, and so on.



Type 3 pipeline computation

The ith process ($0 < i < n$) receives the values x_0, x_1, x_2, …, x_{i-1} and computes x_i from the equation:

$$x_i = \frac{b_i - \sum_{j=0}^{i-1} a_{i,j} x_j}{a_{i,i}}$$

Sequential Code

Given constants $a_{i,j}$ and b_k stored in arrays **a[ ][ ]** and **b[ ]**, respectively, and values for unknowns to be stored in array, **x[ ]**, sequential code could be

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

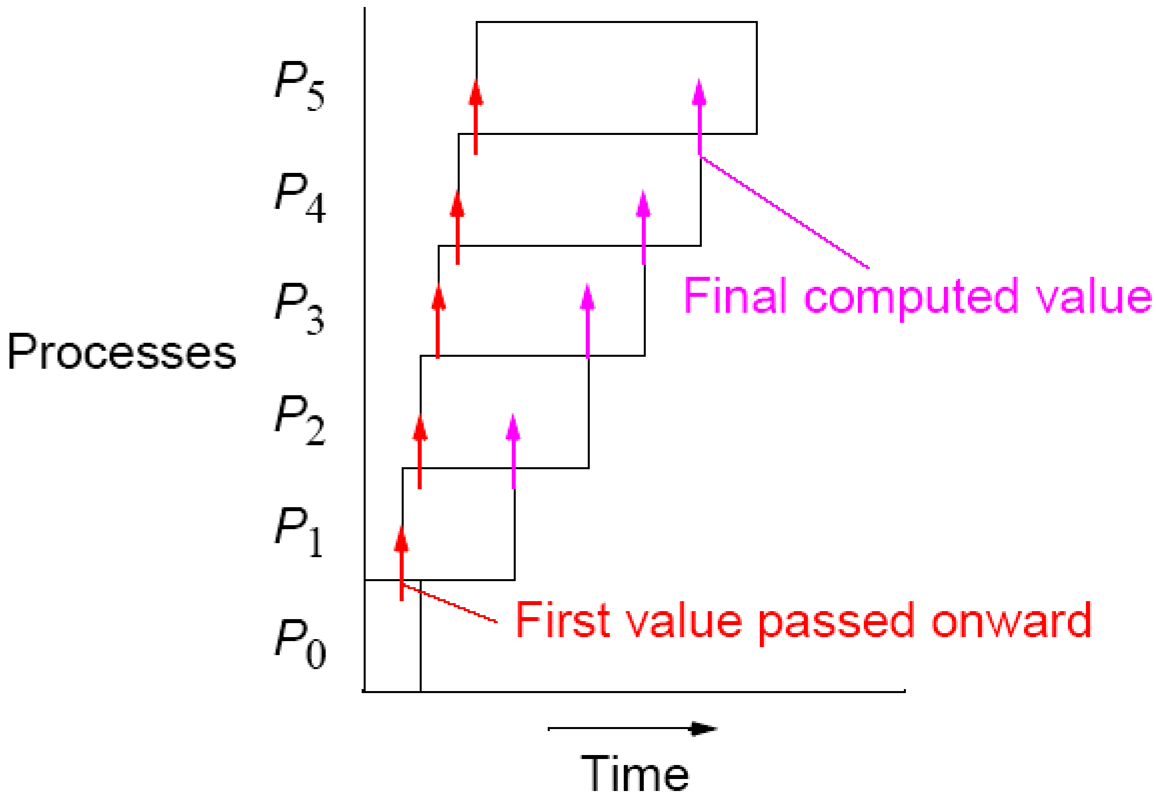
Parallel Code

Pseudocode of process P_i (1 < i < n) of could be

```
for (j = 0; j < i; j++) {
        recv(&x[j], Pi-1);
        send(&x[j], Pi+1);
}
sum = 0;
for (j = 0; j < i; j++)
        sum = sum + a[i][j]*x[j];
x[i] = (b[i] - sum)/a[i][i];
send(&x[i], Pi+1);
```

Now have additional computations to do after receiving and resending values.

Pipeline processing using back substitution



Analysis of Pipelined Form of backsolve.

Multiplications $= t_{mult} \sum_{i=1}^{N} (i-1) + 1$

$= t_{mult} (0.5N(N-1) + N)$

Communications $= \sum_{i=1}^{N} (\mathrm{t}_s + it_{data})$

$= Nt_s + t_{data} 0.5N(N+1)$

PARALLEL BACKSOLVE USING da CUNHA and HOPKINS

- Partition rows of upper triangular matrix into fixed blocks

- Each processor has multiple blocks of rows

- As soon as block of results ready it is distributed to all the other processors.

- As soon as a processor j gets a result x(i) it can use it to perform part of the computation a(j,i)*x(i)/a(j,j)

- Algorithm shows good performance, but what about scalability?

Example of Distribution 2 processors 8 rows array.

Proc	
1	a(7,0)x0+a(7,1)x1+a(7,2)x2+a(7,3)x3+…+a(7,7)x7=b7
1	a(6,0)x0+a(6,1)x1+a(6,2)x2+a(6,3)x3+…+a(6,6)x6=b6
0	a(5,0)x0+a(5,1)x1+a(5,2)x2+a(5,3)x3+…+a(5,5)x5=b5
0	a(4,0)x0+a(4,1)x1+a(4,2)x2+a(4,3)x3+a(4,4)x(4)=b4
1	a(3,0)x0+a(3,1)x1+a(3,2)x2+a(3,3)x3=b3
1	a(2,0)x0+a(2,1)x1+a(2,2)x2=b2
0	a(1,0)x0+a(1,1)x1=b1
0	a(0,0)x0 = b0

Analysis of Distributed Form of backsolve.

Communications N sends to p   Nlog(p)

Multiplications N N/p   (overestimate – assumes all rows are full)

Total Time N log(p) (t_s +t_{data}) + NN/p t_{mult}

Compare against previous shows that there is a Speedup of p against previous multiplications

There is also a speedup in the communications
If N log(p)(t_s +t_{data}) << N t_s + 0.5N(N+1)t_{data}