Overview: Parallelisation via Pipelining

- definition
- pipelining example 1: adding m sets of n numbers each
  - sequential algorithm
  - pipelined parallel algorithm
  - performance analysis
- pipelining example 2: sorting $(p=)n$ numbers
  - sequential insertion sort
  - pipelined parallel insertion sort
  - performance analysis
- pipelining example 3: solving unit lower triangular linear systems with $(p=)n$ equations/unknowns
  - sequential solver
  - pipelined parallel solver

Ref: Wilkinson and Allen Ch 5.

Pipelining

- parallelization technique applicable to a wide range of problems which are partially sequential in nature
- the problem is split into a series of tasks/stages/processes that have to be completed one after another (e.g., in sequence)
- applicable at many levels. For example, at the CPU core level, it is used to extract parallelism out of a single stream of instructions (ILP); other examples of hardware pipelining are systolic array processors (e.g. Google TPUs matrix-matrix multiplier)
- in a message-passing setting, it is realized by a set of processes logically connected as an array or ring operating in "lock-step", where, at every stage, each process receives data from the left processor, and passes onwards data to the right processor, typically after performing some calculations with the data received
- pipelining type of parallelism is not readily apparent in many cases, and one may have to rearrange the order in which the operations are performed in order to expose it (as, e.g., in the solution of triangular systems covered later on)
- harder to understand, and analyze, it may require extremely low-latency interconnection networks to actually lead to significant speed-up in practical scenarios

Example 1: Adding Numbers (Sequential Algorithm)

- let us assume that we have m different sets of n numbers each
- we want to sum the n numbers in each set, for all sets (m different sums)
- the sequential algorithm looks like
  (assume numbers to be summed up are hold in a 2D array `nums[m][n]`):

```
for (i=0; i<m; i++) {
 sums[i]=
   compute_sum(&nums[i][0],n);
}
```

```
function compute_sum(x,n) {
  result=0;
  for (j=0; j<n; j++) {
    result=result+x[j];
  }
  return result;
}
```

- a possible parallelization of this problem might be based on 1D data partitioning by columns + partial/local sums + (all)reduce sum
- how can we parallelize this computation using pipelining? (this lecture)

Example 1: Adding Numbers (pipelined parallelization)

- partition the n numbers in each set across processes, i.e., 1D column-wise partition of `nums[m][n]`
- assume $p=n$, i.e., the pipeline has as many processes as numbers in each set
- the numbers assigned to a given process are stored in a 1D array (`nums_col[m]`)

```
me=process_rank_id();
for (i=0; i<m; i++) {
 sums[i]=
   compute_sum(&nums_col[i],me);
}
```

```
function compute_sum(x,me) {
  me>0 ? recv(&sum,1,me-1) : sum=0;
  sum=sum+x;
  if ((me+1)<p)
   send(&sum,1,me+1);
  return sum;
}
```

- how parallelism is being exploited when the algorithm is executed on a (suitable) parallel computer? (next slide)

Example 1: Adding Numbers (visualizing pipelining parallelism)

$\longrightarrow$ time

P_0	+	S	+	S	+	S	+	S	+	S	+	S	+	...
P_1		R	+	S	R;+	S	R;+	S	R;+	S	R;+	S	R;+	...
P_2				R	+	S	R;+	S	R;+	S	R;+	S	R;+	...
P_3						R	+	R	+	R	+	R	+	...

$\underbrace{\quad}_{\text{cycle1}}$ $\underbrace{\quad}_{\text{cycle2}}$ $\underbrace{\quad}_{\text{cycle3}}$ ...

$\underbrace{\qquad\qquad}_{n-1}$ $\underbrace{\qquad\qquad\qquad}_{m}$

- +: `sum=sum+x;` S: `send(&sum,1,me+1)` R: `recv(&sum,1,me-1)`
- assume *eager* sends (message transfer can proceed without intervention of receiver process) and that the underlying parallel system can provide simultaneous transfers between adjacent processors
- letters in red (next slide)

Example 1: Adding Numbers (visualizing pipelining parallelism)

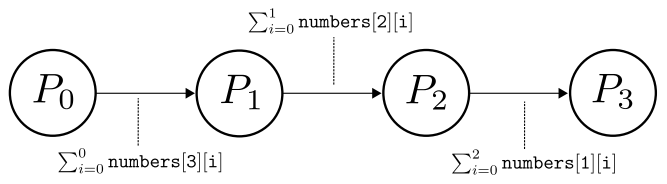
$\longrightarrow$ time

P_0	+	S	+	S	+	S	+	S	+	S	+	S	+	...
P_1		R	+	S	R;+	S	R;+	S	R;+	S	R;+	S	R;+	...
P_2				R	+	S	R;+	S	R;+	S	R;+	S	R;+	...
P_3						R	+	R	+	R	+	R	+	...

$\underbrace{\quad}_{\text{cycle1}}$ $\underbrace{\quad}_{\text{cycle2}}$ $\underbrace{\quad}_{\text{cycle3}}$ ...

$\underbrace{\qquad\qquad}_{n-1}$ $\underbrace{\qquad\qquad\qquad}_{m}$

- letters in red (see figure)


$\sum_{i=0}^{1}$ numbers[2][i]
P_0
P_1
P_2
P_3
$\sum_{i=0}^{0}$ numbers[3][i]
$\sum_{i=0}^{2}$ numbers[1][i]

General Pipeline Performance Analysis

- adding numbers example corresponds to a general class of pipelined algorithms which aim to accelerate the execution of multiple instances of the same problem
- the performance of this sort of parallel pipelined algorithms can be modelled as:
  - assume each process performs similar actions in each pipeline cycle
  - work out computation and communication for a cycle
  - compute the total execution time as:

$$\begin{aligned} t_{\text{total}} &= (\text{time for one pipeline cycle}) * (\text{number of cycles}) \\ &= (t_{\text{comp}} + t_{\text{comm}}) * (m + n - 1) \end{aligned}$$

    where m is the number of instances and n the number of pipeline stages (processes)
  - average time for a computation is then given by $t_{\text{av}} = \frac{t_{\text{total}}}{m}$

Example 1: Adding Numbers (Performance Analysis)

- single instance (i.e., $m = 1$), $p = n$ (i.e., as many processes as numbers per set):

$$\begin{aligned} t_{\text{comp}} &= t_f \\ t_{\text{comm}} &= t_s + t_w \\ t_{\text{total}} &= ((t_s + t_w) + t_f)n \end{aligned}$$

- multiple instances, $p = n$ (i.e., as many processes as numbers per set):

$$\begin{aligned} t_{\text{total}} &= ((t_s + t_w) + t_f)(m + n - 1) \\ t_{\text{av}} &= \frac{t_{\text{total}}}{m} \approx (t_s + t_w) + t_f \text{ for } m \gg n \end{aligned}$$

- multiple instances, data partitioning, i.e., $p = \frac{n}{d}$ (with d data entries per processor)

$$t_{\text{total}} = ((t_s + t_w) + dt_f)(m + \frac{n}{d} - 1)$$

  d influences trade-off among communication and degree of parallelism

Example 2: Insertion Sort (sequential algorithm)

- we want to order a set of n numbers in decreasing order using insertion sort
- the algorithm works with an array of fixed size n, initially with no numbers in it
- for each iteration i, it inserts a new number into the array such that at the end of the iteration the first i numbers inserted are already ordered in decreasing order
- to this end, it first finds the position where to insert the number, and shifts right the remaining ones from that position on
- very poor sequential algorithm, $O(n^2)$ complexity

```
x=memalloc(n);
sz=0; // current array size
for (i=0; i<n; i++) {
  num=rand();
  sz=sort_insert(x,sz,num);
}

function sort_insert(x,sz,num)
  pos=find_position(x,sz,num);
  insert_shift(x,sz,pos,num);
  return sz+1;
```

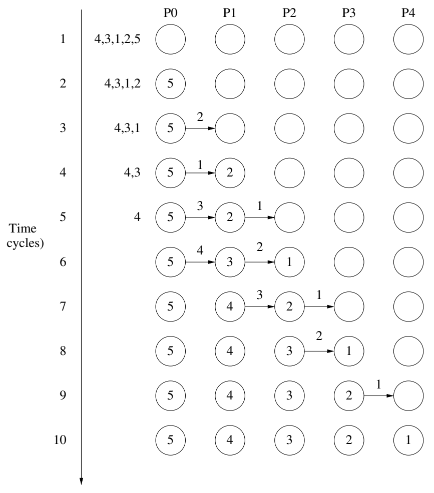
```
function find_position(x,sz,num)
  pos=0;
  while (pos<sz && x[pos]>num)
    pos=pos+1;
  return pos;

function insert_shift(x,sz,pos,num)
  tmp1=x[pos];
  x[pos]=num;
  while (pos<sz)
    tmp2=x[pos+1];x[pos+1]=tmp1;tmp1=tmp2;
    pos=pos+1;
```

Example 2: Insertion Sort (sequential algorithm)

assume we want to insert the numbers 5, 2, 1, 3, and 4

outer loop iteration	array					action
insert 5	5					store 5
insert 2	5					$5 > 2$?
	5	2				store 2
insert 1	5	2				$5 > 1$?
	5	2				$2 > 1$?
	5	2	1			store 1
insert 3	5	2	1			$5 > 3$?
	5	3	1			$2 > 3$? store 3; $2 \rightarrow$
	5	3	2			store 2; $1 \rightarrow$
	5	3	2	1		store 1
insert 4	5	3	2	1		$5 > 4$?
	5	4	2	1		$3 > 4$?; store 4; $3 \rightarrow$
	5	4	3	1		store 3; $2 \rightarrow$
	5	4	3	2		store 2; $1 \rightarrow$
	5	4	3	2	1	store 1

Example 2: Insertion Sort (parallel pipelined algorithm)



- each process stores a single number
- process 0 accepts series of numbers one at a time
- each process in the pipeline keeps the largest number among those that have passed through it (*so that when it does not receive more then it will have the largest among all received*)
- to this end:
  - if number received is smaller than the one stored, then the received number is passed onward (sent) to the next process in the pipeline
  - if not, then the number received replaces the currently stored one, and the latter is passed onwards

Example 2: Insertion Sort (parallel pipelined algorithm)

- key observation: process i receives $n-i$ numbers and passes onwards $n-i-1$

```
me=process_rank_id();
n=num_processes();

// receive (or generate) 1st number and store it
me>0 ? recv(&num,1,me-1) : num=rand();
largest=num

num_procs_to_the_right=n-me-1
for (i=0; i<num_procs_to_the_right; i++) {
  // receive (or generate) next number in the series
  me>0 ? recv(&num,1,me-1) : num=rand();

  if (num>largest) {
    send(&largest,1,me+1); // pass largest number so far onwards
    largest=num;
  }
  else // num <= largest
    send(&num,1,me+1);    // pass received number onwards
}
```

- how parallelism is being exploited when the algorithm is executed on a (suitable) parallel computer? (next slide)

Example 2: Insertion Sort (visualizing pipelined parallelism)

$\longrightarrow$ time

P_0	"R"	"R";>?	S	"R";>?	S	"R";>?	S	"R";>?	S						
P_1			R		R	>?	S	R;>?	S	R;>?	S				
P_2							R		R	>?	S	R;>?	S		
P_3											R		R	>?	S
P_4															R

cycle 1 · cycle 2 . . . cycle $n-1$ . . . cycle $2n-3$

- "R": `num=rand();`
- R: `recv(&num,1,me-1)`
- S: `send(&...,1,me+1)`
- assume *eager* sends (message transfer can proceed without intervention of receiver process) and that the underlying parallel system can provide simultaneous transfers between adjacent processors

Pipelined Insertion Sort Performance Analysis

- sequential:

  $t_s = (n-1)+(n-2)+\cdots+2+1 = \frac{n(n-1)}{2}$

  i.e. $O(n^2)$ – very poor algorithm!
- parallel:
  - each pipeline cycle

    $t_{\text{comp}} = t_c$

    $t_{\text{comm}} = (t_s + t_w)$
  - total execution time (note: $p = n$ here):

    $t_{\text{total}} = (t_{\text{comp}} + t_{\text{comm}})(2n-3) = (t_c + t_s + t_w)(2n-3)$

    i.e. overall $O(n)$ scaling!

Exercise: how can we extend the pipelined insertion sort algorithm so that P_0 receives the series of ordered numbers one at a time after passing onwards his last number at cycle $n-1$? (Hint: think about bidirectional pipeline communication)

Example 3: Unit Lower Triangular Systems (problem definition)

- we aim at finding $x \in \mathbb{R}^n$ such that:

  $$Lx = b$$

  with $L \in \mathbb{R}^{n\times n}$ (a dense nonsingular unit lower triangular matrix) and $b \in \mathbb{R}^n$ given
- in component-wise form, this problem reads (assuming 0-based indexing):

$$\underbrace{\begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{1,0} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n-1,0} & l_{n-1,1} & \cdots & 1 \end{pmatrix}}_{L} \underbrace{\begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{n-1} \end{pmatrix}}_{x} = \underbrace{\begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{pmatrix}}_{b}$$

- n is the number of equations/unknowns in the system
- the solution of this problem is required in a final step when using direct methods (i.e., LU factorization) to solve general linear systems $Ax = b$
- highly scalable parallel implementations available in the public domain, mostly based on 2D block cycling partitioning, though (e.g., MAGMA, ScaLAPACK)

Solving Unit Lower Triangular Systems (forward substitution)

- unit lower triangular systems can be easily solved using forward substitution
  1. solve for x_0 as $x_0 := b_0$;
  2. substitute x_0 into the second equation, and solve for x_1 as $x_1 := b_1 - l_{1,0}x_0$
  3. and so on ...
- each component x_i is obtained by means of the following recurrence:

  $$x_i = b_i - \sum_{j=0}^{i-1} l_{ij}x_j,\ i = 0,\ldots,n-1$$

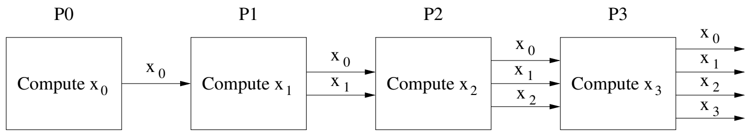
  where x_i depends on the previous $i-1$ components of x
- Sequential code: (one-to-one transcription of previous recurrence)

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

- do you foresee any parallelization challenges of this sequential algorithm?

Forward Substitution: Pipeline Solution I

- consider a row-wise partition of L and a static mapping of a single row per process, i.e., $p = n$ (i.e., we have as many rows as processes)
- the vector b is partitioned/mapped accordingly to the rows of L
- the vector x is replicated in all processes
- a first attempt to pipelining could look like as (see figure below):
  - process 0 computes x_0 and sends its value to the next process
  - in general, process i receives the values $x_0, x_1, \dots, x_{i-1}$ from processor $i-1$ and computes x_i; then, it sends $x_0, x_1, \dots, x_i$ to process $i+1$



- what's wrong with this pipelined algorithm?

First "Pipelined" Algorithm for Forward Substitution

- := : `x[i]=sum;`
- $*$: `l[i][j]*x[j]`
- $-$: `sum = sum - ...;`

$\longrightarrow$ time

P_0	:=	S																	
P_1		R	$*$	$-$	:=	S													
P_2						R	$*$	$-$	$*$	$-$	:=	S							
P_3												R	$*$	$-$	$*$	$-$	$*$	$-$	:=

- P_3 gets x_0 very late, but *it was already available at the first step!*
- how can we reorder the steps in the pipelined algorithm such that we eagerly foster that data arrives (i.e., that data dependencies are satisfied) as soon as possible?

Forward Substitution: Pipeline Solution II

```
me=process_rank_id();
n=num_processes();
sum = b[me];
for (j = 1; j < me; j++) {
  recv(&x[j], 1, me-1);
  if (me+1 < n) send(&x[j], 1, me+1);
  sum = sum - a[i][j]*x[j];
}
if (me+1 < n) send(&sum, 1, me+1);
x[me] = sum
```

"Pipelined" Algorithm for Forward Substitution

- := : `x[i]=sum;`
- $*$: `l[i][j]*x[j]`
- $-$: `sum = sum - ...;`

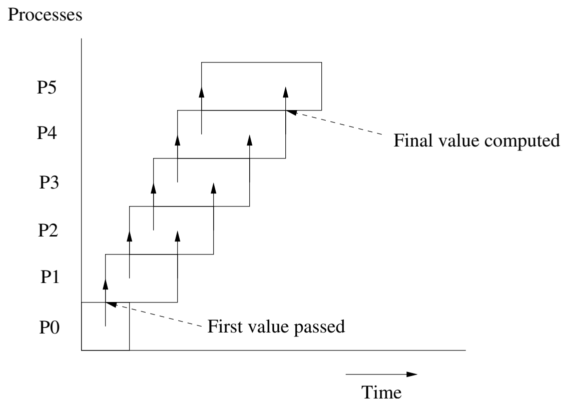
$\longrightarrow$ time

P_0	:=	S																	
P_1		R	S	$*$	$-$	S	:=												
P_2			R	S	$*$	$-$	R	S	$*$	$-$	S	:=							
P_3				R	S	$*$	$-$	R	S	$*$	$-$	R	S	$*$	$-$	S	:=		
$\vdots$					$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

- assuming a perfect synchronization of sends/recvs as above, the parallel execution time will be given by the the cost of $n-1$ data transfers plus the time spent by the last processor to calculate x_{n-1} once it receives the first message

General Class of Pipelined Problems (Time Diagram)

- the pipelined algorithm for forward substitution actually belongs to a general class of pipelined algorithms in which the information to start the next process can be passed forward before the process itself has completed all its internal operations


Processes
P5
P4
P3
P2
P1
P0
Final value computed
First value passed
Time