Parallel Algorithms

Sequential vs. Parallel Algorithms

Sequences are one of the most fundamental aspects of computer science

Algorithms and computer programs were initially conceived and formulated according to the concept of sequences.

As computer science matured, it became apparent that many fundamental operations in algorithms could occur simultaneously (i.e. in parallel) leading to the development of the parallel computer (i.e., a computer with more than one processor)

There are currently two major architectural approaches to parallel computing.

Shared memory computers (usually called multiprocessors) give all of the individual processors access to a common shared memory (data structures and values in memory are shared between the processors).

Message passing computers (usually called multicomputers) give each processor its own local memory and processors share data by passing messages over some type of computer network.

Parallel algorithms place more complex requirements on software.

An algorithm for a sequential computer provides a sequence of operations for a single processor.

An algorithm for a parallel computer provides a sequence of operations for each processor to follow in parallel, including operations that co-ordinate and integrate the individual processors into one coherent task.

Parallel Algorithms

A process is the basic building block of a parallel algorithm.

Informally, a process is a subroutine or procedure that is executed by a single, specific physical processor.

A powerful and common technique for organizing parallel algorithms is data parallelism.

Data parallelism results in the same operation being performed on every component of a data structure in parallel where the data structure resides in shared memory.

Data parallelism makes used of a parallel form of an iterative loop, where all iterations of the loop are performed in parallel.

We will assume the use of a `forall` statement for creating parallel processes in an iterative loop.

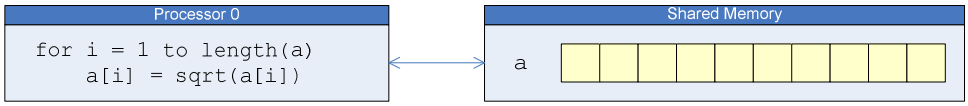
`forall` is a parallel form of the `for` loop in which all of the loop iterations are executed in parallel rather than sequentially.

Example

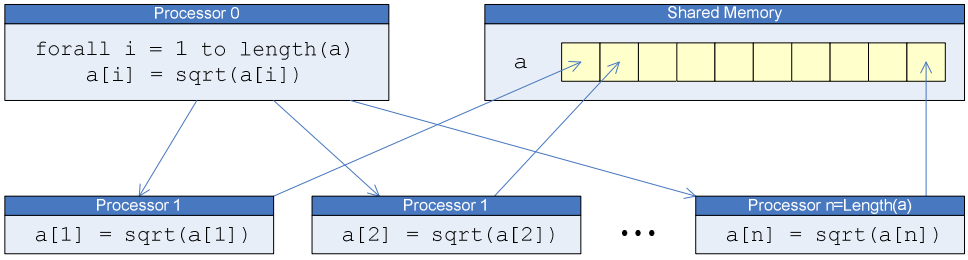
```
for i = 1 to length(a)
   a[i] = sqrt(a[i])

forall i = 1 to length(a)
   a[i] = sqrt(a[i])
```

Sequential Process Creation



Parallel Process Creation



Speedup Example

Sequential

```
for i = 1 to 100
   [Code Body]
```

Time	*Action*
10	Process created
10,010	First iteration done
20,010	Second iteration done
30,010	Third iteration done
...	
990,010	99^{th} iteration done
10,00,010	100^{th} iteration done

Parallel

```
forall i = 1 to 100
   [Code Body]
```

Time	*Action*
10	Process 1 created
20	Process 2 created
30	Process 3 created
...	
990	Process 99 created
1000	Process 100 created
10,010	Process 1 done
10,020	Process 2 done
10,030	Process 3 done
...	
10,990	Process 99 done
11,000	Process 100 done

$$SpeedUp = \frac{Sequential\ Time}{Parallel\ Time} = \frac{1{,}000{,}010}{11{,}000} = 90.91 \approx 100$$

This is a rather simplified example. The performance of parallel programs can be limited by:

Memory contention

Excessive sequential code

Process creation time

Communication delay

Synchronization delay

Load imbalance

Parallel Sorting Example

Rank sort is a simple parallel sorting algorithm where each element of an array is compared with every other element of the array to see which is larger.

The rank of an element is defined as the total number of elements less than the element.

The final position of an element in the sorted array is just its rank.

Sequential rank sort algorithm

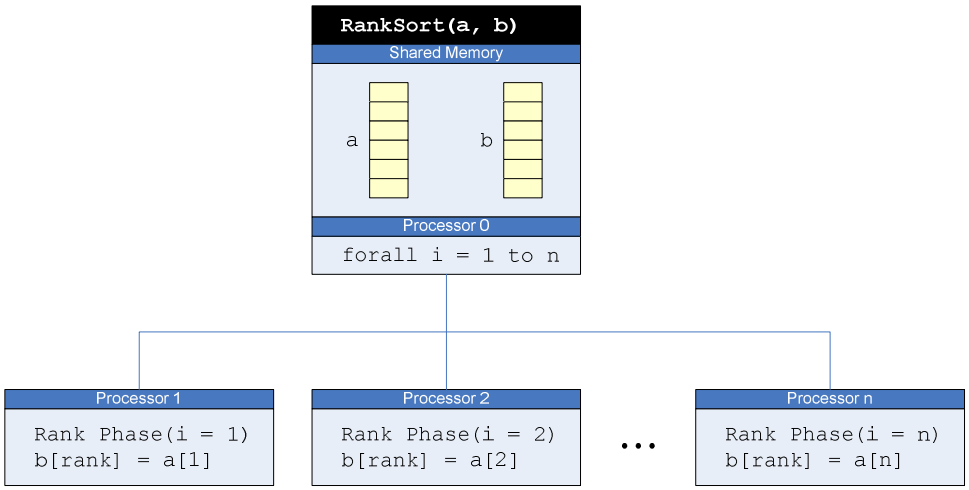
```
SequentialRankSort(a, b)

n = Length(a)
for i = 1 to n
   rank = 1
   for j = 1 to i - 1
      if a[j] <= a[i]
         rank++
   for j = i + 1 to n
      if a[j] < a[i]
         rank++
   b[rank] = a[i]
```

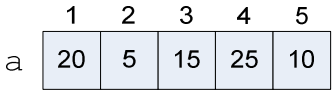
Parallel rank sort algorithm

```
ParallelRankSort(a, b)

n = Length(a)
forall i = 1 to n
   rank = 1
   for j = 1 to i - 1
      if a[j] <= a[i]
         rank++
   for j = i + 1 to n
      if a[j] < a[i]
         rank++
   b[rank] = a[i]
```

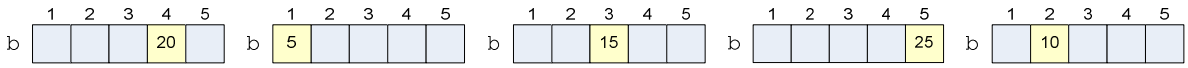


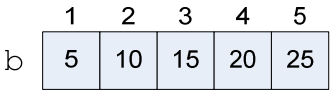
Example



Processor 1	Processor 2	Processor 3	Processor 4	Processor 5
`i = 1`	`i = 2`	`i = 3`	`i = 4`	`i = 5`

`rank = 1`
`j = 2:`
`  rank++ (2)`
`j = 3:`
`  rank++ (3)`
`j = 4:`
`j = 5:`
`  rank++ (4)`
`b[4] = a[1]`
	`rank = 1`
	`j = 1:`
	`j = 3:`
	`j = 4:`
	`j = 5:`
	`b[1] = a[2]`
		`rank = 1`
		`j = 1:`
		`j = 2:`
		`  rank++ (2)`
		`j = 4:`
		`j = 5:`
		`  rank++ (3)`
		`b[3] = a[3]`
			`rank = 1`
			`j = 1:`
			`  rank++ (2)`
			`j = 2:`
			`  rank++ (3)`
			`j = 3:`
			`  rank++ (4)`
			`j = 5:`
			`  rank++ (5)`
			`b[5] = a[4]`
				`rank = 1`
				`j = 1:`
				`j = 2:`
				`  rank++ (2)`
				`j = 3:`
				`j = 4:`
				`b[2] = a[5]`





Since all processors are executing in parallel, the total run time is $O(n)$. This is faster than the fastest known sequential sorting algorithms which are $O(n \log n)$.

If the number of physical processors P is less than the number of array elements n, then the algorithm should be modified to assign $\frac{n}{P}$ elements to each processor.

$P = 100$ processors

$n = 1000$ elements

So, assign $\frac{1000}{100} = 10$ elements to each processor.