3 Partitioning and Divide-and-Conquer Strategies

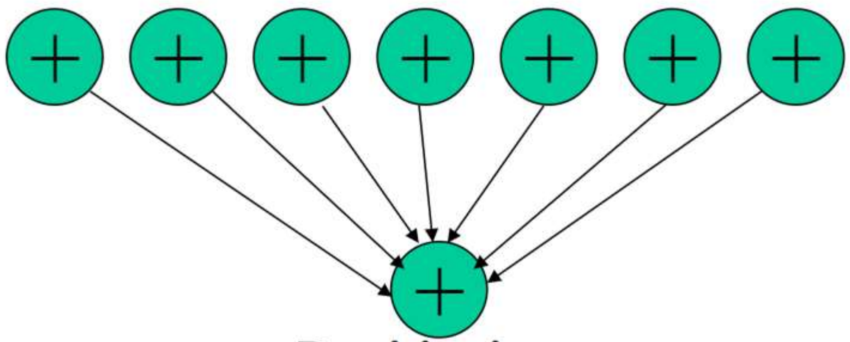
[Weightage(10%): Approx. 7 Marks out of 70 Marks]

- Partitioning
- Divide-and-Conquer

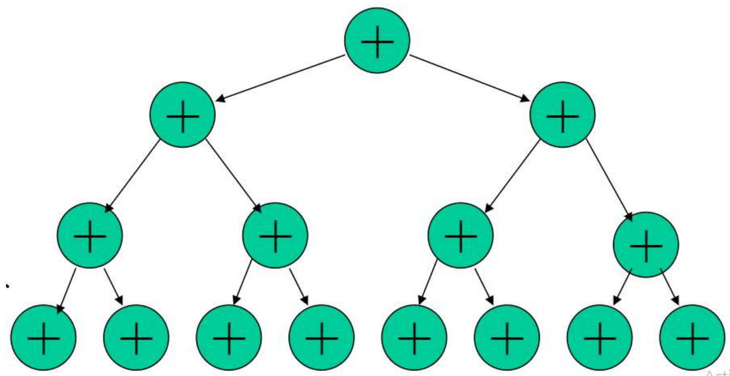
Strategies

3.1 Partitioning: The problem is simply divided into separate parts and each part is computed separately.

3.2 Divide & Conquer: It usually applies partitioning in a recursive manner by continually dividing the problem into smaller and smaller parts before solving the smaller parts and combining the results.



Partitioning



Divide and Conquer

Image Source: https://slideplayer.com/slide/4497115/

3.1 Partitioning

Partitioning Strategies:

Data partitioning (Domain decomposition): Partitioning can be applied to the program data (that is, to dividing the data and operating upon the divided data concurrently).

Functional decomposition: Partitioning can also be applied to the functions of a program (that is, dividing the program into independent functions and executing the functions concurrently).

It is much less common to find concurrent functions in a problem, but data partitioning is a main strategy for parallel programming.

3.1 Partitioning

Example: Adding 'n' numbers. (E.g. **n**=20)
X0, X1, X2, X3, X4, X5, X6, X7, X8, X9, X10, X11, X12, X13, X14, X15, X16, X17, X18, X19, X20

Let us consider dividing the problem into 'm' parts (E.g. **m**=4)
m=1: $X_0+X_1+X_2+X_3+X_4$ [PartialSum_1]
m=2: $X_5+X_6+X_7+X_8+X_9$ [PartialSum_2]
m=3: $X_{10}+X_{11}+X_{12}+X_{13}+X_{14}$ [PartialSum_3]
m=4: $X_{16}+X_{17}+X_{18}+X_{19}+X_{20}$ [PartialSum_4]

Sum = $\text{PartialSum}_1+\ldots+\text{PartialSum}_m$

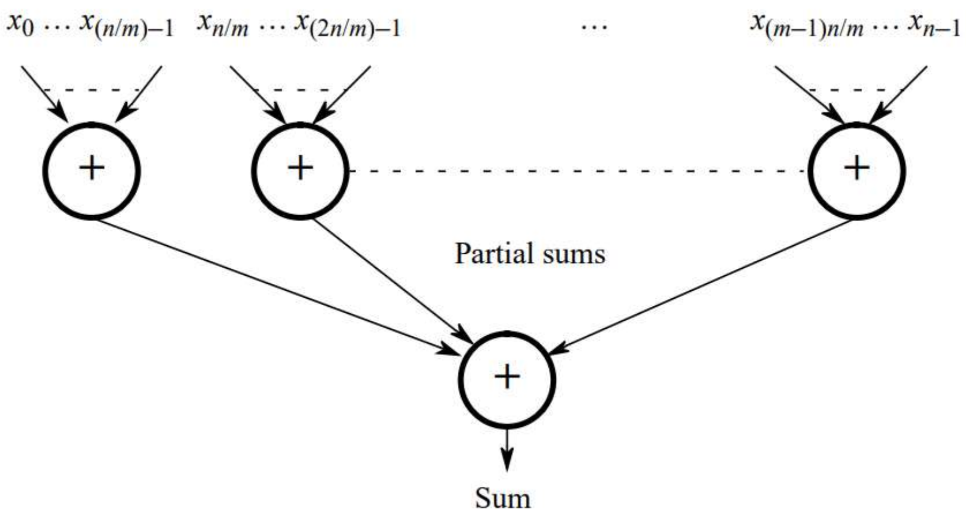


Figure 4.1 Partitioning a sequence of numbers into parts and adding the parts.

3.1 Partitioning (Simple)

Master

```
s = n/m;                                    /* number of numbers for slaves*/
for (i = 0, x = 0; i < m; i++, x = x + s)
   send(&numbers[x], s, Pi);                /* send s numbers to slave */

result = 0;
for (i = 0; i < m; i++) {                   /* wait for results from slaves */
   recv(&part_sum, PANY);
   sum = sum + part_sum;                    /* accumulate partial sums */
}
```

Slave

```
recv(numbers, s, Pmaster);                  /* receive s numbers from master */
sum = 0;
for (i = 0; i < s; i++)                     /* add numbers */
   part_sum = part_sum + numbers[i];
send(&part_sum, Pmaster);                   /* send sum to master */
```

3.1 Partitioning (Broadcast)

Master

```
s = n/m;                                  /* number of numbers for slaves */
bcast(numbers, s, P_slave_group);         /* send all numbers to slaves */
result = 0;
for (i = 0; i < m; i++){                  /* wait for results from slaves */
   recv(&part_sum, P_ANY);
   sum = sum + part_sum;                  /* accumulate partial sums */
}
```

Slave

```
bcast(numbers, s, P_master);              /* receive all numbers from master*/
start = slave_number * s;                 /* slave number obtained earlier */
end = start + s;
sum = 0;
for (i = start; i < end; i++)             /* add numbers */
   part_sum = part_sum + numbers[i];
send(&part_sum, P_master);                /* send sum to master */
```

3.1 Partitioning (Scatter/Gather)

Master

```
s = n/m;                                          /* number of numbers */
scatter(numbers,&s,P_group,root=master);          /* send numbers to slaves */
reduce_add(&sum,&s,P_group,root=master);          /* results from slaves */
```

Slave

```
scatter(numbers,&s,P_group,root=master);          /* receive s numbers */
reduce_add(&part_sum,&s,P_group,root=master);/* send sum to master */
```

3.1 Partitioning (Phases: Communication / Computation)

Phase 1 — Communication: First, we need to consider the communication aspect of the m slave processes reading their n/m numbers.
Using individual send and receive routines requires a communication time of $t_{comm1} = m(t_{startup} + (n/m)t_{data})$
where $t_{startup}$ is the constant time portion of the transmission and t_{data} is the time to transmit one data word. Using scatter might reduce the number of startup times; i.e., $t_{comm1} = t_{startup} + nt_{data}$

Phase 2 — Computation: Next, we need to estimate the number of computational steps. The slave processes each add n/m numbers together, requiring $n/m - 1$ additions.
Since all m slave processes are operating together, we can consider all the partial sums obtained in the $n/m - 1$ steps. Hence, the parallel computation time of this phase is $t_{comp1} = n/m - 1$

3.2 Divide-and-Conquer

- The divide-and-conquer approach is characterized by dividing a problem into sub problems that are of the **same form** as the larger problem.
- Further divisions into still **smaller sub problems** are usually done by **recursion**, a method well known to sequential programmers.
- The recursive method will continually divide a problem until the tasks **cannot be broken** down into smaller parts.
- Then the very simple tasks are performed and **results combined**, with the combining continued with larger and larger tasks

3.2 Divide-and-Conquer

```
int add(int *s) /* add list of numbers, s */
     {
     if (number(s) =< 2) return (n1 + n2);
     else {
          Divide (s, s1, s2);
          part_sum1 = add(s1);
          part_sum2 = add(s2);
          return (part_sum1 + part_sum2);
          }
     }
```

3.2 Divide-and-Conquer (Binary tree construction)

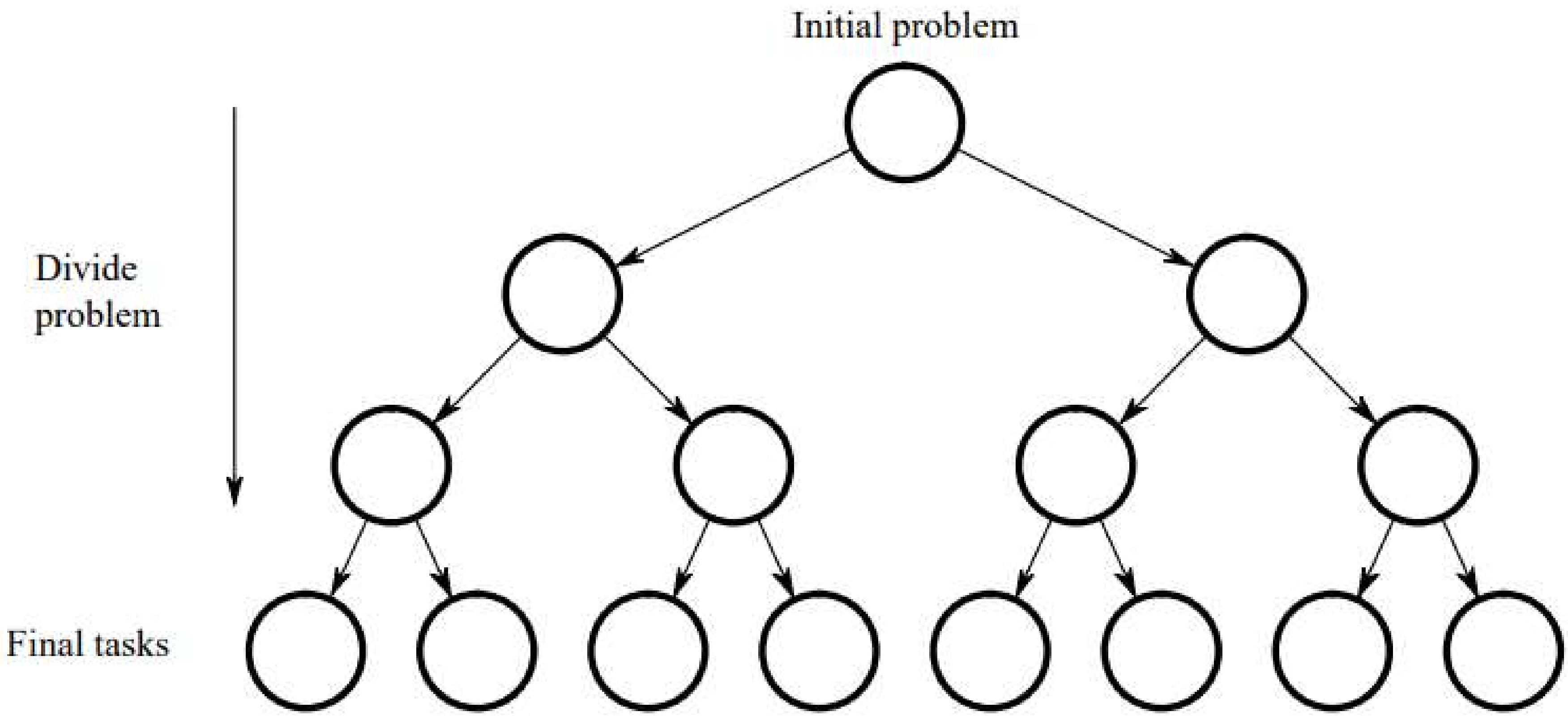


Figure 4.2 Tree construction.

3.2 Divide-and-Conquer (Binary tree construction)

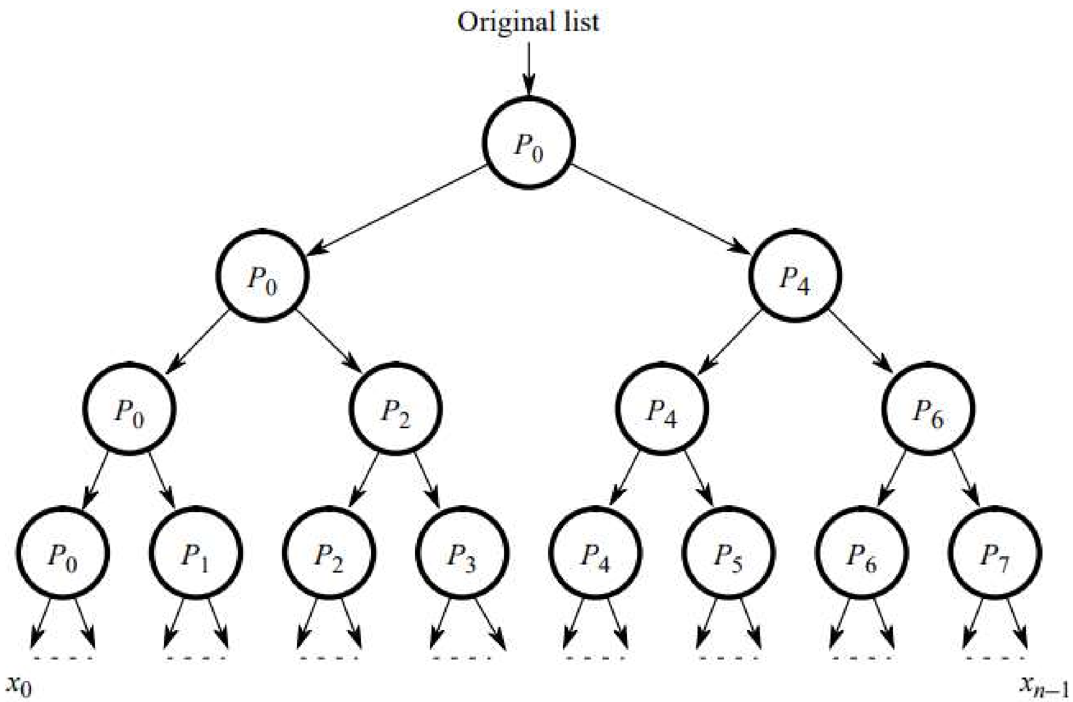


Figure 4.3 Dividing a list into parts.

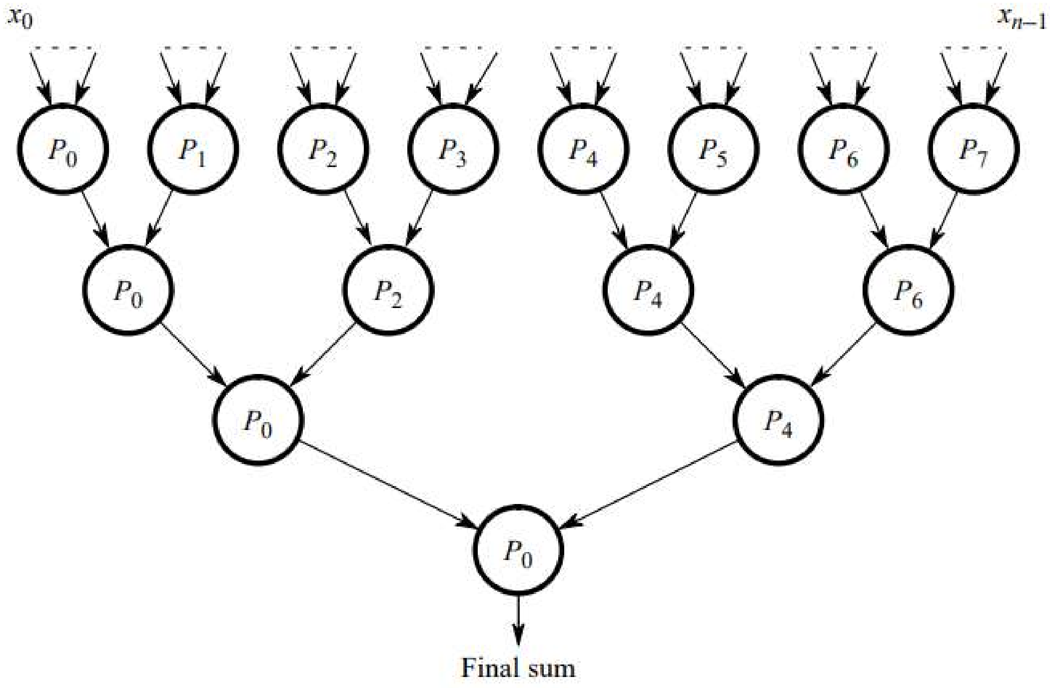


Figure 4.4 Partial summation.

3.2 Divide-and-Conquer (Binary tree construction)

Process P0

```
/* division phase */
divide(s1, s1, s2);
/* divide s1 into two, s1 and s2 */
send(s2, P4);
/* send one part to another process */
divide(s1, s1, s2);
send(s2, P2);
divide(s1, s1, s2);
send(s2, P0};
part_sum = *s1; /* combining phase */
recv(&part_sum1, P0);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P2);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P4);
part_sum = part_sum + part_sum1;
```

Process P4

```
/* division phase */
recv(s1, P0);
divide(s1, s1, s2);
send(s2, P6);
divide(s1, s1, s2);
send(s2, P5);
part_sum = *s1;
/* combining phase */
recv(&part_sum1, P5);
part_sum = part_sum + part_sum1;
recv(&part_sum1, P6);
part_sum = part_sum + part_sum1;
send(&part_sum, P0);
```

3.2 M-ary Divide and Conquer

Divide and conquer can also be applied where a task is divided into **more than two parts** at each stage. For example, if the task is broken into four parts, the sequential recursive definition would be

```
int add(int *s)
      {
      if (number(s) =< 4) return(n1 + n2 + n3 + n4);
      else {
            Divide (s,s1,s2,s3,s4);
            part_sum1 = add(s1);
            part_sum2 = add(s2);
            part_sum3 = add(s3);
            part_sum4 = add(s4);
            return (part_sum1 + part_sum2 + part_sum3 + part_sum4);
            }
      }
```

3 Partitioning and Divide-and-Conquer Strategies

[Weightage(10%): Approx. 7 Marks out of 70 Marks]

1. List the partitioning strategies and explain them in brief.
2. Explain the partitioning with the example of adding 'n' numbers.
3. Explain the communication and computation phases of partitioning.
4. Explain divide-and-conquer in brief.
5. Write the algorithm for divide-and-conquer.
6. Explain M-ary Divide and Conquer.