Section 12.1

1) If we assume that one person can add two numbers in ta time, how long will it take that person to add two n×n matrices, if we consider the operation of addition as the basic operation? Justify your answer.

It would take one person t_a*n^2 time to add two n × n matrices, since the addition of a pair of elements would have to be done once for each position in the resulting matrix sequentially.

2) If we have two people add numbers and it takes ta time for one person to add two numbers, how long will the two people take to add two n×n matrices, if we consider the operation of addition as the basic operation? Justify your answer.

It would take two people $(t_a*n^2)/2$ time to add two n × n matrices. It would take half the amount of time as it would for one person, since none of the values either person computes depends on the values the other person computes.

3) Consider the problem of adding two n×n matrices. If it takes ta time for one person to add two numbers, how many people do we need to minimize the total time spent to get the final answer? What will be the minimum amount of time needed to find the answer, if we assume that we have enough people? Justify your answers.

Having n^2 people adding two n × n matrices would minimize the time for performing the addition. With n^2 people, it would take just one time step (t_a time), since none of the values are dependent on the results of others. More people would not be needed since they could not reduce the time any further.

4) Assuming that one person can add two numbers in ta time, how long will it take that person to add all n numbers of a list, if we consider the operation of addition as the basic operation? Justify your answer.

It would take one person $t_a*(n-1)$ time to add all the numbers in a list of size n, since after the addition of the first two elements they would have to add the current value of the running sum to the next element in the list sequentially.

5) If we have two people add n numbers in a list and it takes ta time for one person to add two numbers, how long will it take the two people to add all n numbers in the list, if we consider the operation of addition as the basic operation and include tp time for passing the result of an addition from one person to the other? Justify your answer.

It would take two people $t_a*(n/2)$ time to add all the numbers in a list of size n, since each person must add up half the list in the same manner as Exercise 4 (n/2 – 1 additions), and then one of them must make the final addition of both of their results at the end.

6) Consider the problem of adding n numbers in a list. If it takes ta time for one person to add two numbers and it takes no time to pass the result of an addition from one person to another, how many people do we need to minimize the total time spent to get the final answer? What will be the minimum amount of time needed to find the answer, if we assume we have enough people? Justify your answer.

Having n/2 people adding a list of n numbers would minimize the time for performing the addition. With n/2 people, it would take $t_a*\log n$ time since the sum of each pair of numbers added by one person in the first iteration could be added to the result of another person's calculation by each of half as many people in the second iteration, and so on, reducing the number of additions by half at each iteration. More people would not reduce the time any further since the result of each addition operation is necessary to compute the result of the next addition.

Section 12.2

7) Write a CREW PRAM algorithm for adding all n numbers in a list in $\Theta(\lg n)$ time.

Problem: Find the sum of all n numbers in a list.

Inputs: positive integer n, array of numbers S[] indexed from 1 to n.

Outputs: the sum of all the numbers in S.

Comment: It is assumed that n is a power of 2 and that we have n/2 processors indexed from 1 to n/2.

```
number sum(int n, number S[]) {
    index step, size = 1;
    local index p = index of this processor;
```

```
    local number num1, num2;
    for(step = 1; step <= log n; step++) {
        if((p-1) % size == 0) {
            read S[2*p – 1] into num1;
            read S[2*p – 1 + size] into num2;
            write num1 + num2 into S[2*p – 1];
            size = size * 2;
        }
    }
    return S[1];
}
```

The loop runs for log n steps for the first processor and so the complexity is $\Theta(\log n)$.

8) Write a CREW PRAM algorithm that uses n^2 processors to multiply two n×n matrices. Your algorithm should perform better than the standard $\Theta(n^3)$-time serial algorithm.

Problem: Multiply two $n \times n$ matrices.

Inputs: positive integer n, two $n \times n$ arrays of numbers, S[][] and T[][].

Outputs: An $n \times n$ array of numbers U that is product of S and T.

Comments: We assume there are n^2 processors available, each with a unique pair of indices in $\{1, 2, \ldots, n\}^2$.

```
void matMult(int n, number S[][], number T[][], number& U[][]) {
    local index i, j, k;
    local int a, b;
    i = the first index of this processor;
    j = the second index of this processor;
    local int sum = 0;
    for(k=1; k<=n; k++) {
        read S[i][k] into a;
        read T[k][j] into b;
        sum = sum + a*b;
    }
    write sum into U[i][j];
```

}

Each processor computes a sum of n numbers and writes the result to the position in the U array to which is corresponds. Computing this sum takes O(n) time since it must loop through n numbers to do so. Thus, the whole operation takes O(n) time with n^2 processors.

9) Write a PRAM algorithm for Quicksort using n processors to sort a list of n elements.

```
void parQuicksort(int N, index left, index right, keytype S[], keytype T[]) {
        keytype pivot = S[left];
        bool small;
        local index p;   local element;
        index crt_left = left;  index crt_right = right;
        p = index of this processor;
        read S[p] into element;
        if (element < pivot)
                small = True;
        else
                small = False;
        for (i = 1; i <= n; i++)
                if (small) {
                        write element into T[crt_left];
                        crt_left++;
                }
                else {
                        write element into T[crt_right];
                        crt_right++;
                }
        //at this point, crt_left + 1 equals crt_right – 1
        Write pivot into T[crt_right+1];
        parQuicksort(crt_left – left + 1,     left,         crt_left, S[], T[]);
        parQuicksort(right – crt_right + 1, crt_right,  right,     S[], T[])
```

```
}
```

10) Write a sequential algorithm that implements the Tournament method to find the largest key in an array of n keys. Show that this algorithm is no more efficient than the standard sequential algorithm.

```
for (step = 1; step <= lg n; step = 2*step)
        for (i = 1; i <= n; i  = i + step*2) {
                a1 = S[i];
                a2 = S[i+step];
                if (a1 < a2)
                        S[i] = a2; //largest goes in left member of the pair
        }
//At the end, S[1] will store the largest element.
```

Analysis: In the first iteration of the outer for loop, there are n/2 comparisons, in the second iteration n/4, then n/8, … , until we reach 1 comparison in the last round. The total is n/2 + n/4 + n/8 + … + 1 = n-1, exactly as in the standard sequential algorithm developed in Ch.1.

11) Write a PRAM algorithm using n^3 processors to multiply two n×n matrices. Your algorithm should run in $\Theta(\lg n)$ time.

Problem: Multiply two n × n matrices.

Inputs: positive integer n, two n × n arrays of numbers, S[][] and T[][].

Outputs: An n × n array of numbers U that is product of S and T.

Comments: We assume there are n^3 processors available, each with a unique triple of indices in $\{1, 2, \ldots, n\}^3$.

```
void matMult(int n, number S[][], number T[][], number& U[][]) {
    local index i, j, k;
    local int a, b;
    i = the first index of this processor;
    j = the second index of this processor;
    k = the third index of this processor;
```

```
    int step, size = 2;
    int V[1…n][1…n][1…n];
    read S[i][k] into a;
    read T[k][j] into b;
    write a*b into V[i][j][k];
    for(step=1; step<=log n; step++) {
            if((k-1)%size==0) {
            read V[i][j][2*k – 1] into a;
            read V[i][j][2*k – 1 +size] into b;
            write a + b into S[2*k – 1];
            size = size *2;
        }
}
    write V[i][j][1] into U[i][j];
}
```

Each processor first writes the product of two numbers associated with its indices to its corresponding spot in array V. This takes constant time with n^3 processors. The loop is then executed to compute the sum of the list of elements in V[i][j] for each pair of indices i and j. This sum can be computed in log n time using n/2 processors for each sum. Thus, only half of the processors are needed for this step. The method is the same as that used in Exercise 7, except that size starts out with value 2, meaning that half of the processors never do anything during the loop. The loop thus takes log n time and the whole operation takes O(log n) time with n^3 processors.

12) Write a PRAM algorithm for the Shortest Paths problem of Section3.2. Compare the performance of your algorithm against the performance of Floyd's algorithm (Algorithm 3.3).

We use n^2 processors to implement the inner two for loops in parallel:

```
void parFloyd(int n, number D[][]) {
    local index i, j, k;
    local int a, b, c;
    i = the first index of this processor;
    j = the second index of this processor;
    for (k = 1; k <= n; k++) {
```

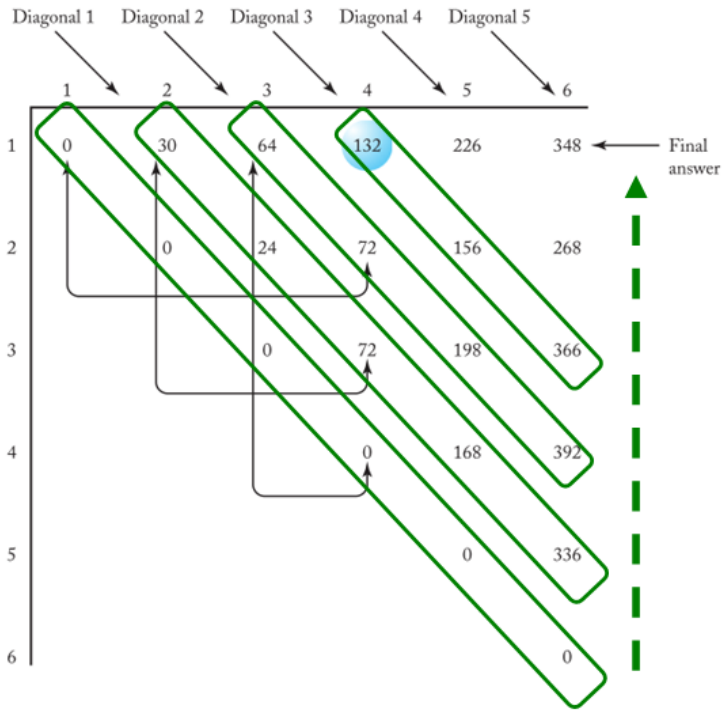
```
            read D[i][j] into a;
            read D[i][k] into b;
            read D[k][j] into c;
            a = minimum(a, b + c);
            write a into D[i][j];
        }
```

Analysis: Since there is only one for loop, this algorithm is Θ(n).

13) Write a PRAM algorithm for the Chained Matrix Multiplication problem of Section 3.4. Compare the performance of your algorithm against the performance of the Minimum Multiplications algorithm (Algorithm 3.6).

Refer to Figure 3.8 for explanations:

Figure 3.8
The array *M* developed in Example 3.5. *M*[1][4], which is circled, is computed from the pairs of entries indicated.



The parallel algorithm will proceed in a loop with n iterations, implementing first the calculations in the main diagonal of the matrix, then the secondary diagonal above it, and so on until the upper-right corner. Note that initially n processors are needed, but their number decreases by one in each iteration, until the upper-left corner only requires one processor.

```
int parCMM(int n, const int d[], int M[][]) {
        index diagonal;
        local index p = index of this processor;
        local min, i, j, k, a, b, c, d, e;
        write 0 to M[p][p];                    //zeroes in parallel on main diagonal
        for (diagonal=1; diagonal<=n-1; diagonal++)
                if (p <= n–diagonal+1) {       //if processor p is being used in this iteration
                        i = p; j = p + diagonal -1;     //coordinates of the position that processor
                                                        //p is updating in this iteration
                        min = large number;   //calculate the minimum
                        for (k=1; k<=j-1; k++) {
                                read M[i][k] into a;    read M[k+1][j] into b;
                                read d[i-1] into c;     read d[k] into d;       read d[j] into e;
                                if (a + b + c*d*e < min)
                                        min = a + b + c*d*e;
                        }
                        Write min into M[i][j];
                } //end if
        return M[1][n];
}
```

For simplicity, the algorithm above does not calculate P[i][j], the value that gives the minimum. The reader is encouraged to complete the code.

Analysis: Iteration *diagonal* of the outer for loop requires calculating the minimum of k elements, which is done with k – 1 comparisons, so the every-case time complexity is $\Theta(0+1+2+\ldots+n-1) = \Theta(n^2)$, compared to $\Theta(n^3)$ for the serial algorithm.

14) Write a PRAM algorithm for the Optimal Binary Search Tree problem of Section 3.5. Compare the performance of your algorithm against the performance of the Optimal Binary Search Tree algorithm (Algorithm 3.9).

```
float parOST() {int n, const float p[]) {
        index diagonal;
        float A[1..n+1][0..n];
        local min, i, j, k, a, b, c;
        local index p = index of this processor;
        for (i=1; i<=n; i++) {              //initialize the boundary elements
                A[i][i-1] = 0;
                A[i][i] = p[i];
        }
        A[n+1][n] = 0;
        for (diagonal=1; diagonal<=n-1; diagonal++)
                if (p <= n–diagonal+1) {        //if processor p is being used in this iteration
                        i = p; j = p + diagonal;        //coordinates of the position that processor
                                                        //p is updating in this iteration
                        min = large number;   //calculate the minimum
                        for (k=i; k<=j; k++) {
                                read A[i][k-1] into a;   read A[k+1][j] into b;
                                read [sum of p(m) for i<=m<=j] into c;
                                if (a + b + c < min)
                                        min = a + b + c;
                        }
                        Write min into A[i][j];
                } //end if
        return A[1][n];
}
```

For simplicity, the algorithm above does <u>not</u>:

- Show how to efficiently calculate **[sum of p(m) for i<=m<=j]**. Take a look at Exercise 23 in Ch.3 to remember how it's done (dynamic programming!)
- Calculate R[i][j], the index that gives the minimum. The reader is encouraged to complete the code.

Analysis: Iteration *diagonal* of the outer for loop requires calculating the minimum of k elements, which is done with k – 1 comparisons, so the every-case time complexity is $\Theta(0+1+2+\ldots+n-1) = \Theta(n^2)$, compared to $\Theta(n^3)$ for the serial algorithm.

Additional Exercises

15) Consider the problem of adding the numbers in a list of n numbers. If it takes ta(n−1) time for one person to add all n numbers, is it possible for m people to compute the sum in less than [ta(n−1)]/m time? Justify your answer.

No. This is not possible because the result of additions performed by one person must be combined at the next time step with the result of additions performed by another person. Thus, even if there were enough people (n/2) to add each pair of numbers in one step, the sums of each of those pairs would then have to be added, and so on. The lower bound on the time to add n numbers is O(log n), which can be achieved when m = n/2. This would not produce a time cost better than $t_a(n-1)/m$ since $t_a*(n/2)/(n/2) = t_a < t_a*\log n$. Adding more people cannot make the operation any faster.

16. Write a PRAM algorithm that runs in $\Theta((\lg n)^2)$ time for the problem of mergesorting. (Hint: Use n processors, and assign each processor to a key to determine the position of the key in the final list by binary searching.)

The algorithm is called *odd-even sorting* or *bitonic mergesort*, and it was published by Batcher in 1968 (K. E. Batcher, "Sorting Networks and their applications", Proc. AFIPS Spring Joint Summer Computer Conference, 32:307-314, 1968.)

```
void parMergesort(int n, intA[]) {
        local index p = index of this processor;        //between 0 and n-1
        local index counterpart;
        for (k=2; k<=N; k=2*k)
                for (j=k/2; j>0; j=j/2) {
                        counterpart = p^j;
                        if (counterpart > p) {
                                if ((p&k)==0 && A[i] > A[counterpart])
                                        exchange(p, counterpart, A);
```

```
                    if ((p&k)!=0 && A[p] < A[counterpart])
                        exchange(p, counterpart, A);
                } //end if
            } //end inner for
}
void exchange(index i, index j, int A[]) {
    int temp;
    temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
```

Analysis: The outer for loop (on k) iterates lg n times, and, in each iteration, the inner for loop iterates lg k times, so the total complexity is $1 + 2 + 3 + \ldots + \lg n = (\lg n)\cdot(\lg n + 1)/2 \in Q((\lg n)^2)$.

17. Write a PRAM algorithm for the Traveling Salesperson problem of Section 3.6. Compare the performance of your algorithm against the performance of the Traveling Salesperson algorithm (Algorithm 3.11).

For reference, we reproduce below the pseudocode for Algorithm 3.11:

```
void travel (int n, const number W[][], index P[][],
             number& minlength)
{
  index i, j, k; number D[1..n][subset of V - {v1}];

  for (i = 2; i <= n; i++)
     D[i][∅] = W[i][1];
  for (k = 1; k <= n - 2; k++)
     for (all subsets A ⊆ V - {v1} containing k vertices)
        for (i such that i ≠ 1 and vi is not in A){
           D[i][A] = minimum (W[i][j] + D[j][A - {vj}]);
                     j:vj ∈ A
           P[i][A] = value of j that gave the minimum;
        }
  D[1][V - {v1}] = minimum (W[1][j] + D[j][V - {v1,vj}]);
                   2 ≤ j ≤ n
  P[1][V - {v1}] = value of j that gave the minimum;
  minlength = D[1][V - {v1}];
}
```

We implement the outer for loop (on k) in parallel, on n – 2 processors, by distributing each value of k to a processor p; each processor executes in parallel the block made up of the inner two loops.

```
void parTravel (int n, const W[][], index P[][], number& minlength) {
        local index i, j, k, a, b;
        local index p = index of this processor;        //p between 1 and n – 2
        number D[1..n][subset of V – {v1}];
        for (i=2; i<=n; i++)
            D[i][empty set] = W[i][1];
        k = p;
        for (all subsets A ⊆ V – {v1} containing k vertices)
            for (i such that i ≠ 1 and v_i is not in A){
                a = minimum (W[i][j] + D[j][A – {v_j}]);
                    j:v_j ∈ A
                b = value of j that gave the minimum;
                write a into D[i][A];
                write b into P[i][A];
            }
        D[1][V – {v1}] = minimum (W[1][j] + D[j][V – {v1, v_j}]);
                         2 ≤ j ≤ n
        P[1][V – {v1}] = value of j that gave the minimum;
        minlength = D[1][V – {v1}];
}
```

Since we have split the task into $\Theta(n)$ parallel tasks, our algorithm is $\Theta(n2^n)$, compared to $\Theta(n^2 2^n)$ for the serial Algorithm 3.11.