

CHAPTER 6. APPLICATIONS.

In this chapter we consider several applications of the principles you have studied in the previous chapters. We consider here parallelization of several algorithms that are quite well known in their serial version. The algorithms we consider are not always the best suited to the shared memory architecture of Pluto. As often happens in parallel programming, some algorithms discussed here are better suited to different architectures. Nevertheless, they provide us important illustrations.

6.1 Merging

Merging refers to the process of combining two sorted lists $X = \{x_0, x_1, \dots, x_r\}$ and $Y = \{y_0, y_1, \dots, y_{s-1}\}$ into a single sorted list $Z = \{z_0, z_1, \dots, z_{r+s-1}\}$. The usual requirements are that as sets, $Z = X \chi Y$, if X, Y are nondecreasing, so is Z , and that there is no repetition of elements in Z .

However, we will allow duplicate elements in our merging algorithm. Merging arises in a variety of contexts in sorting, file management, and database applications. Usually, the lists X and Y are sublists of a given list and often arise during sorting the original list. In the Merge Sort algorithm, for example, an essential step in the sorting process is the merging of the sorted halves of a given list. Merging is very well understood in sequential algorithms and the following is a typical algorithm.

Let us suppose that X and Y are two sequences of numbers, sorted in nondecreasing order. To avoid cumbersome subscript notation, we will use $X[i]$ to denote the elements of X , and similarly for Y and Z . We assume that both X and Y are in nondecreasing order. The serial algorithm to merge X and Y is as follows.

We use two pointers one for each sequence. The algorithm uses the following step repeatedly. We compare the elements referenced by the two pointers. The smaller of the two values is assigned to the next element of Z . For example, if the pointers are currently pointing to $X[i]$ and $Y[j]$ respectively, and $Z[0], Z[1], \dots, Z[p-1]$ have already been assigned, the smaller of $X[i]$ and $Y[j]$ is assigned to $Z[p]$. The pointer to the sequence from which $Z[p]$ came is advanced to the next position. Initially, the pointers are positioned at $X[0]$ and $Y[0]$ respectively.

Eventually, one sequence will be finished. The remaining elements of the other sequence are now copied to Z . In pseudocode:

```
void SerialMerge (int *X, int *Y, int *Z) {
    int i=0,j=0,k;

    for (k=0; (i<r)&&(j<s); k++) {
        if (X[i]<Y[j]) {
            Z[k]=X[i];
            i++;
        } else {
            Z[k]=Y[j];
            j++;
        }
    }
    // special cases to handle end of arrays
    while (i<r) {
        Z[k]=X[i];
        i++;
        k++;
    }
    while (j<s) {
        Z[k]=Y[j];
        j++;
        k++;
    }
}
```

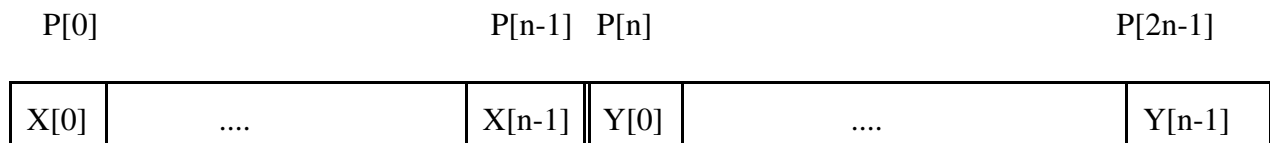
In the worst case when $r = s = n$, the running time of this algorithm is $O(n)$ since it takes at least one comparison to get an element of Z .

This algorithm may seem inherently sequential, and impossible of parallelism. However, there is an ingenious trick. Assume that there are $2n$ processors, numbered $P[0], P[1], \dots, P[2n-1]$. First consider the array X . The job of the processors $P[i]$ is to find the position of $X[i]$, $i = 0, 1, \dots, n-1$, in the final list Z . This is done as follows.

Consider $X[i]$. Then there are (i) elements before it in the array X . Suppose we can find its position in the array Y . It may be the first, last or somewhere between. Assume that we have found $Y[j]$ in Y such that $Y[j-1] < X[i] < Y[j]$. Hence there must be $(j - 1)$ elements before $X[i]$ in the array Y . Hence, in all there are $(i - 1 + j)$ elements before it in the combined array Z . This means that $X[i]$ should be in the $(i + j)$ -th position in Z ! The only question that remains is: "How do we find the position of $X[i]$ in Y ?" The answer: Binary Search. Usually, Binary Search is a function. It is used to find out if a given Key is found in an array A . If the Key is found, it returns the index of the array A where it is found, else zero. Obviously, one would have to modify Binary Search for our intended use here.

How about elements of Y ? We treat them the same way. Processor $P[n+j]$ finds the position of $Y[j]$ in the array X , and then computing its position in Z , $j = 0, 1, \dots, n - 1$.

Duplicate elements, that is, elements that occur in both arrays would pose a problem, since they would both collide at the same spot in Z . To avoid this, we introduce the following asymmetry. When searching for $X[i]$ in Y , we consider any element equal to $X[i]$ as *larger* than $X[i]$. On the other hand, when binary searching for $Y[j]$ in X , any element equal to $Y[j]$ will be treated as *smaller* than $Y[j]$. The following diagram illustrates our algorithm.



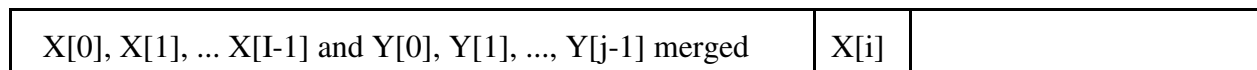
(a) Processor assignments

P[i]



(b) Binary Search: P[i] finds j such that $Y[j+1] < X[i] < Y[j]$

P[i]



Z[i+j+1]

(c) P[i] writes X[i] into Z [i + j]

Pseudocode for parallel merging of two sorted lists is given below.

```

int modified_bsearch (int *list, int len, int key, int weight) {
    int upper, lower, middle;
    // divide list into upper and lower halves
    upper=len-1; lower=0;
    if (list[0]>key) return 0; // special cases, key is outside of list
    if (list[upper]<key) return upper;
    while ( lower+1<upper ) {
        if (list[middle]==key) return middle+weight;
        if (list[middle]<key) lower=middle;// look at upper half
        else upper=middle; // look at lower half
    }
    // list[lower] < key < list[upper] hopefully
    // must handle when list[lower]==key or list[upper]==key]
    return upper;
}
    
```

```

void ParallelMerge(int *X, int *Y, int *Z) {
    int tmp;
    for id = 0 to 2n-1 // do in parallel
        if (id < n) {
            tmp = modified_bsearch(X,r,X[id],0);
            Z[tmp+id]=X[id];
        } else {
            tmp = modified_bsearch(Y,s,Y[id-n],1);
            Z[tmp+id]=Y[id];
        }
    }
}

```

How fast is our algorithm for parallel merging? The initialization of Key is one step. The binary searches are done in lists of n keys, and are done in parallel. Therefore, they take $\log \lambda n \mu$ steps. Writing to Z can be done in the last binary search step. Therefore, the total number of steps is $\log \lambda n \mu + 1$. Since the serial algorithm takes n steps, the speedup is $n / (\log \lambda n \mu + 1)$. Of course, we have made a lavish assumption on the number of processors available! If we have fewer number of physical processors, the work must be divided among them, and the speedup reduced accordingly.

6.2 Sorting

Sorting and searching are important activities in computer science. The elementary sorting algorithms such as Bubble Sort and Insertion Sort are quadratic algorithms, since their complexity is of the order $O(n^2)$ for a list of size n . On the other hand, the better known sorting algorithms such as Quick Sort and Merge Sort, best known in their recursive forms, are of complexity order $O(n \log n)$. It is well known that this is about the best possible for methods that sorting by comparison of values. These are the most used serial sorting algorithms. For lists with fewer than 50 elements, a slow sorting method such as Bubble Sort may be acceptable. For

larger lists, quadratic algorithms are considered much too slow. Nevertheless, in the environment of parallel processing, even quadratic algorithms can become stellar performers. Sorting algorithms on parallel computers, with various architectures, has been well studied. Here, we will consider here several simple parallel sorting methods.

Rank Sort: The principle behind Rank Sort is simple. Suppose that we have to sort a list of n distinct numbers. For each number x in the list, define its *rank* $R(x)$ as the number of elements of the list (excluding x) that are less than x . For example, consider the following list with six elements:

10 15 4 2 13 22

Then $R(10) = 2$, $R(15) = 4$, $R(4) = 1$, $R(2) = 0$, $R(13) = 3$, and $R(22) = 5$. Once the rank is determined, we know exactly where each element belongs in the sorted list. It is precisely.

$R(x)$. The sorted list for our example is: 2 4 10 13 15 22. In the serial version of rank sort algorithm given below, we use this modification.

```
// MAX == length of L
void SerialRankSort(int *L) {
    int tmp[MAX],i,j,rank;

    for (i=0;i<MAX;i++) {
        rank=0;
        for (j=0;j<MAX;j++) {
            if (L[i]>L[j]) rank++;
        }
        tmp[rank]=L[i];
    }
    memcpy(L,tmp,sizeof(int)*MAX);
}
```

To find the complexity of this algorithm, note that to find the rank of a given list element, we have to scan the whole list, that is, n comparisons. This is done for every element of the list.

Therefore the complexity is $O(n^2)$.

This algorithm provides plenty of opportunities to parallelize. Assume that we have n processes. Then ranks of the n elements of the list can be found by the n processes in parallel. This is the simplest parallelization, which essentially parallelizes the outer loop. Each process still has to make n comparisons, although in parallel. The sorted array can be filled in parallel also. Therefore, the complexity of the algorithm is $O(n)$. Notice that this is better than $O(n \log n)$ complexity for the most popular serial algorithms. If we have the luxury of many more processes, then we can also parallelize the inner loop as well, as we did for nested loops in Chapter 5. This will improve the complexity even further. The pseudocode for the parallel version of this algorithm is given below. The original list L and the sorted list $Sorted$ are assumed to be shared.

```
// MAX == length of L and Sorted
void ParallelRankSort(int *L, int *Sorted) {
    int i,rank;

    for id=0 to MAX-1 , do in parallel
        for (i=0;i<MAX;i++) {
            if (L[id]>L[i]) rank++;
        }
        Sorted[rank]=L[id];
    }
}
```

Note that this algorithm makes n^2 comparisons.

Remarks: In the above code, we assumed there are n processes with each process determining the rank of an element giving a speed up of $(n^2/n = n)$. If the number of processes is $p < n$, then we have to use the loopsplitting technique to compute the rank of an element giving a speed up of $(n^2 / (pn/pk * n))$. If the number of processes $p > n$, say $k * n = p$, then (as discussed in Chapter 5) we can divide the available number of processes into groups of k processes each so that both the

loops are parallelized. That procedure is applicable here as well giving a speed up of $(n^2/\phi n/k\kappa)$.

Odd-Even sort: Recall the principle behind Bubble Sort. If A is an array of n elements, then in Bubble Sort, we make several *passes* through the list. In each pass, starting from the beginning, one compares each element of A with its right neighbor (except the last element). If the elements are out of order, they are switched. The effect of this is that at the end of the first pass, the largest element moves into the last place. At the end of the second pass, the second largest element moves into the $(n - 2)$ th position. In general, after the i -th pass, the $(n - i)$ th largest element moves into the $(n - 1 - i)$ th place. It is easy to see that even in the worst case, no more than n passes are needed to sort the array.. The pseudocode that accomplishes this is as follows:

```
// MAX == length of L
void BubbleSort ( int *L ) {
    int i,j;
    for (i=0; i<(n-1);i++)
        for (j=0;j<i;j++)
            if (L[j]>L[j+1]) swap(L[j],L[j+1]);
}
```

This is not the most efficient form of coding this algorithm since it does not recognize a list that is already sorted. However, this will do for purposes of illustration.

Apparently we cannot parallelize this by simply doing the loops in parallel since the values in the inner loop are dependent on the values of the counter in the outer loop. However, there is a clever procedure and this is the principle behind *Odd-Even Sort*. The single pass we make in Bubble Sort is broken into *two steps*. In the first step, we compare elements in *odd* places with their right neighbor, that is we compare the elements at (1, 2), (3, 4),...etc. If not in order, they are switched. In the second step, we do the same thing with elements in *even*

positions, elements at (0, 1), (2, 3),.. etc. After $\lceil n/2 \rceil$ passes through the list, it will be sorted.

We will first present the algorithm in pseudocode, and then illustrate with an example.

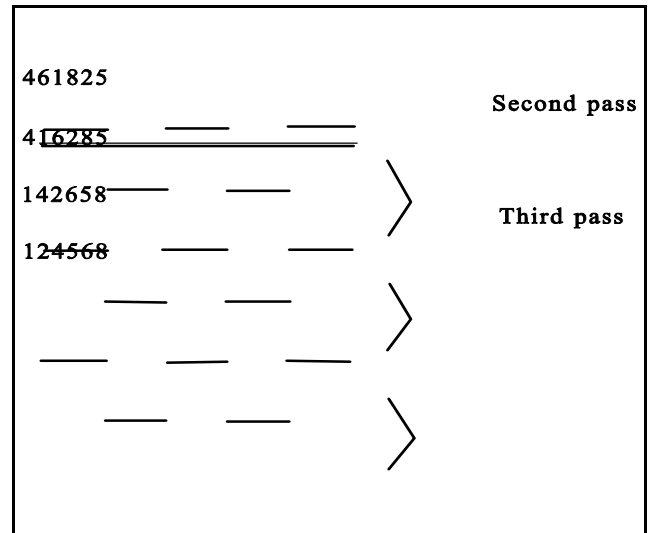
```

// MAX == length of L
void SerialOddEvenSort (int *L) {
    int i,j;
    for (i=0; i <  $\lceil n/2 \rceil$ ; i++) {
        j=1;           // odd step
        do {
            if (L[j]>L[j+1]) swap(L[j], L[j+1]);
            j=j+2;
        } while (j<MAX);
        j=0;           // even step
        do {
            if (L[j]>L[j+1]) swap(L[j], L[j+1]);
            j=j+2;
        } while (j<MAX);
    }
}

```

We illustrate the algorithm with the following example:

The given sequence is 8 6 5 4 1 2. The brackets indicate the three passes. The second line in each bracket shows the status of the sequence after the pass. You should observe several things. By the very nature of the algorithm, the smaller elements move left, the larger ones to the right. Also, the order of the two steps of the algorithm could be reversed. To see



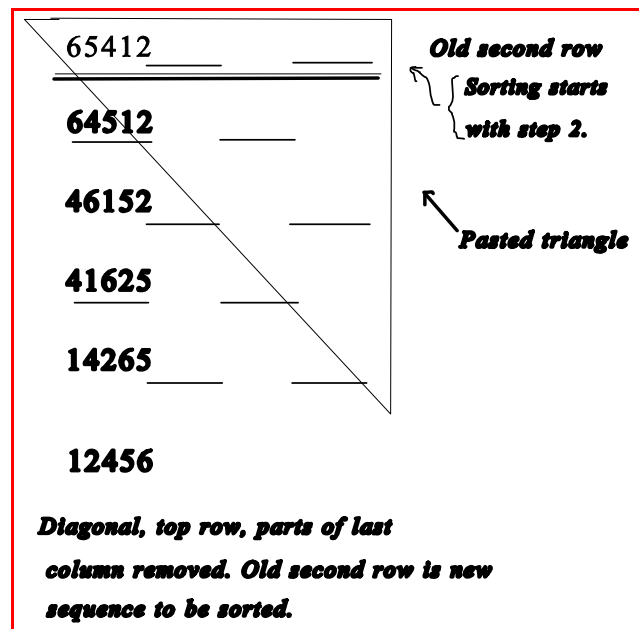
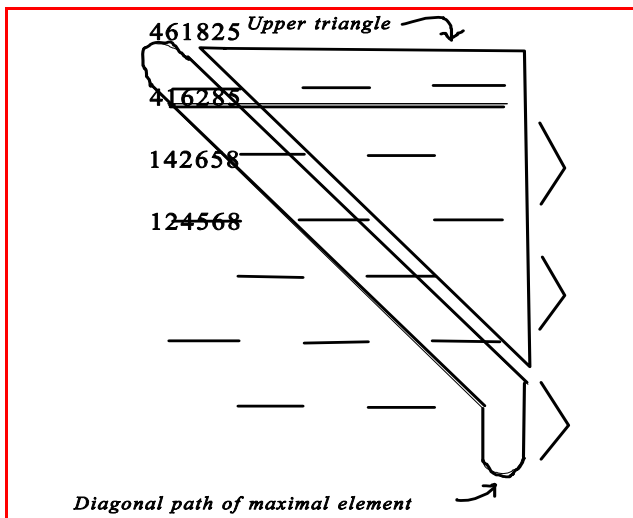
this, imagine a phantom element, smaller than all the other elements in position one. In our example, think of 0 in position one. This element never moves and has no effect on the sorting

of the other elements.. Sorting on this new list starting with step one, is equivalent to sorting on the original list starting with step two. Also notice that we did not have to go into step in the third pass, and that it is already sorted. In fact one can prove the following theorem. However, its proof is less trivial than that of Bubble Sort. We will refer to the above diagram as the *sorting diagram*.

Theorem: *The Odd-Even sorts a given sequence of n elements in at most n steps.*

Proof: The proof is by induction. Clearly, the theorem is true for $n = 2$. Either the two elements are already sorted, or else they will get sorted just by applying the first step. Assume then that the theorem is true for $n = k$. Let A be any other sequence with $(k + 1)$ elements.

Without loss of generality we can assume that the maximal element M of A is in an *odd* position. If not, we can add an extra phantom element, smaller than the minimal element of A , at the beginning. This places M in an odd position. Since it never moves, the phantom element has no effect on the sorting since. Construct the sorting diagram for A and stop after $(k + 1)$ steps. In this diagram trace the path of M . It moves diagonally to the right, coming to rest in the last column, and then stays there.



Suppose that M is in the $(2i + 1)$ -th position. Then there $2i$ elements preceding it. Since they form complete pairs, when step one of the sorting process is applied to the first row, they continue to occupy the first $2i$ positions. Let the element after M be denoted by N . Currently N is in position $(2i + 2)$.

Now remove M (in our example $M = 8$) from the sort diagram completely by removing it from its diagonal path, and also from the last column. Notice that this creates a triangular piece in the right top corner, and a truncated last column. Cut the triangular piece and paste it over the sort diagram so that

1. the *first row* of the triangular piece is over the *second row* of the sorting diagram,
2. N is over itself in the second row (the last column is now full), and
3. remove the truncated first row.

The new first row is a new sequence of k elements. Its first $2i$ elements are the same (some possibly in different positions) as those in the old sequence. Because of the removal of M from the original sequence, and the fact that the new sequence is a result of applying step one to the old sequence, *the second row in the new sort diagram is obtained by applying step two to the new sequence*. Since M only moves to the right by changing its position with its right neighbor, *it does not affect the sorting process of the other elements*. (In the example above, observe the diagram in the left. Read the second row, jumping over 8 (the maximal element). You get the second row of the new diagram on the right.)

It follows therefore that the new sort diagram is obtained by applying the algorithm with

steps one and two reversed.

There are now only $(k-1)$ elements but k rows. By induction hypotheses, the new sequence must be sorted by now. By simply adding M at the end to the last row, we obtain a sorting of the old given sequence. However, we removed a row. Hence the old sequence is sorted in $(k + 1)$ steps as was to be proved. \square

Parallel Odd-Even Sort: It is fairly obvious how the serial version of the Odd-Even sort must be parallelized. The pseudocode is given below. We have assumed that $\lfloor n/2 \rfloor$ processes are available.:

```
// MAX == length of L
void ParallelOddEvenSort ( int *L ) {
    int i,j;
    for id = 0 to  $\lfloor n/2 \rfloor$  do in parallel
        for (i=0;i< $\lfloor n/2 \rfloor$ ; i++) {
            j=1+2*id;          // odd step
            if ((j+1)<n) {
                if (List[j]>List[j+1]) swap(L[j],L[j+1]);
            }
            j=2*id;          // even step
            if ((j+1)<n) {
                if (List[j]>List[j+1]) swap(L[j],L[j+1]);
            }
        }
    }
}
```

How fast is this algorithm? During one pass through the sequence, we perform both steps. Each of the steps does one comparison, and possibly one switch. This is done in constant time. By the theorem we need no more than n steps in all. Hence the complexity is $O(n)$. Clearly the speedup is n . With fewer processors, the speedup will suffer.

Odd-Even Merge Sort: This is an interesting combination of merging, we considered in the beginning of this chapter, and Odd-Even sort. We will only briefly describe the algorithm but will not give a proof of its correctness. The idea is this: Given n elements, say $n = 100$, we break it into several small, and roughly equal, sublists. Suppose we divide it into 5 sublists, and number them 1 through 5. Then each sublist has 20 elements. Each sublist is sorted in parallel using a fast serial sorting algorithm such as Merge Sort or Quick Sort. At this point, each of the sublists 1 through five is sorted.

We now use parallel merging and Odd-Even Sort in combination. Groups 1 and 2 are first merged to form a list of 40 elements. They are broken into two groups again, with the first half in group 1 and the second half in group 2. We do the same for Groups 3 and 4. This corresponds to step one in Odd-Even Sort. In fact, if each group has exactly one element, this is Odd-Even Sort.

In the next step, we merge Groups 2 and 3 into a single group, and then break them again into two groups. Again, the first half is placed in group 2 and the second half in group 3.

We repeat this procedure of merging the groups, and breaking them again 5 times (equal to the number of groups). At the end of the procedure, the original sequence is sorted. The algorithm is quite fast! If n is divided into p groups, each group has n/p elements. Each group is sorted in parallel, and the serial sorting time for this is $O((n/p) \log (n/p))$. Using serial merge, merging the two lists into one of size $(2n/p)$ is of complexity at most $O(2n/p)$, that is $O(n/p)$. Breaking this merged list into their respective two groups is also of order $O(n/p)$, since we may have to copy elements from the merged list into the groups. Thus each of the two steps of Odd-Even Sort applied here takes $O(n/p)$ time units.

We do this in a loop, $\frac{n}{p}$ times. Hence the final complexity estimate is,

$$\left(\frac{n}{p}\right) \cdot \{O(n/p)\log(n/p)\} + O(n/p) = O((n\log n)/p) + O(n).$$

For example, with $n = 1000$, $p = 10$, the total time units is about $(1000 \times 10)/10 + 1000 = 2000$.

Since the best serial sorting algorithm has complexity $O(n\log n)$, speedup = 5. If we had parallel merge, we would have even better speedup.

6.3 Solution of linear systems

Since many matrix operations can be parallelized easily, the solution of a system of n equations in as many unknowns, lends itself to parallel computation. Solution of linear systems is a necessary step in the solution of engineering and scientific problems. In this section, we will consider two methods of solving a system of linear equations by two methods. The first is the well known method of *Gaussian* elimination. This method ends in a finite number of steps. The second is an *iterative* method. There are several such methods, but we will consider the simplest of them, the *Jacobi* method. This method produces a sequence of approximations to the solution and is terminated when it satisfies a termination criteria.

1. Gaussian elimination

The method is best described by means of an example. Consider the following system of 3 equations in 3 unknowns, x_1 , x_2 and x_3 :

$$x_1 + 2x_2 + x_3 = 0 \quad (1)$$

$$2x_1 + 2x_2 + 3x_3 = 3 \quad (2)$$

$$-x_1 - 3x_2 = 2 \quad (3)$$

Step 1: Determine the equation that has numerically the largest coefficient of x_1 . This is the *pivot*, and the row containing it the *pivot row*. If the pivot row is not the first equation, switch the first equation with the pivot row. In our example, the pivot row is equation (2). We switch it with the first equation and renumber the equations.

$$2x_1 + 2x_2 + 3x_3 = 3 \quad (4)$$

$$x_1 + 2x_2 + x_3 = 0 \quad (5)$$

$$-x_1 - 3x_2 = 2 \quad (6)$$

Step 2: Using equation (4), eliminate x_1 from equations (5) and (6). In the case of equation (5), we do this by multiplying equation (4) by -2 and adding it to equation (5). For equation (6), the multiplier is 2 and we multiply equation (4) by this number and add to (6). The coefficients of x_1 in equations (5) and (6) are now zero. The transformed equations are:

$$2x_1 + 2x_2 + 3x_3 = 3 \quad (7)$$

$$0x_1 + x_2 - 2x_3 = -3/2 \quad (8)$$

$$0x_1 - 2x_2 + 3/2x_3 = 7/2 \quad (9)$$

Step 3: Equation (7) will no longer be considered except at the end. We focus on the *reduced system of equations* (8) and (9). We now repeat steps one and two for equation (8). We first find the pivot which now is the numerically largest coefficient of x_2 . This occurs in equation (9) which is the pivot row. We exchange rows and (8) and (9) to get the new system:

$$2x_1 + 2x_2 + 3x_3 = 3 \quad (10)$$

$$0x_1 - 2x_2 + 3/2x_3 = 7/2 \quad (11)$$

$$0x_1 + x_2 - 2x_3 = -3/2 \quad (12)$$

Using (11), this time we eliminate x_2 from equation (12), which is the only one left.

The multiplier now is 2. We multiply (11) by 2 and add to (12). The transformed system is:

$$2x_1 + 2x_2 + 3x_3 = 3 \quad (13)$$

$$0x_1 - 2x_2 + 3/2x_3 = 7/2 \quad (14)$$

$$0x_1 + 0x_2 + 3x_3 = 3 \quad (15)$$

Step 4: The elimination of x_1 and x_2 transforms the original system into *upper triangular form*. We solve this system by *back substitution*. From (15), $x_3 = 1$. Substituting this in (14), we get $x_2 = -1$. Substituting both of this in (13) we $x_1 = 1$.

Let us summarize our procedure. In matrix notation, we started with a linear system of equations of the form $Ax = b$. We converted into an equivalent upper triangular system of the form $Ux = c$, by adding multiples of one equation (the pivot row) to another equation. This resulted in the elimination of some variables in the second equation. Finally, we solved the system $Ux = c$ by back substitution.

Gaussian elimination algorithm: Let the given linear system be written as:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \dots + a_{1j}x_j + \dots + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \dots + a_{2j}x_j + \dots + a_{2n}x_n = b_2$$

.....

$$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \dots + a_{nj}x_j + \dots + a_{1n}x_n = b_n$$

From the example presented, it is clear that we successively eliminate $x_1, x_2, x_3, \dots, x_j, \dots, x_n$ from the second through last equation. At the k -th stage ($1 \leq k \leq n$), we eliminate x_k from the remaining equations as follows:

- a. Find the pivot $A_{p,k} : A_{p,k} = \max \{ |A_{i,k}| : k \leq i \leq n \}$
- b. Exchange the k -th and p -th rows
- c. Eliminate x_k from equation i , ($k+1 \leq i \leq n$) by doing the following:

Multiply the k -th equation by the multiplier $M_{i,k} = A_{i,k} / A_{k,k}$ and add to equation i .

This transforms the other coefficients $A_{i,j}$ as follows:

$$A_{i,j} \leftarrow A_{i,j} - A_{i,k} \cdot A_{k,j} / A_{k,k}$$

At this point, the original system $Ax = b$ has been transformed into the system $Ux = c$. To solve for x , we use back substitution. From the last equation find x_n :

$$x_n = (c_n - u_{n,n} x_n) / u_{n,n}$$

Substituting this in the $(n-1)$ -st equation,

$$u_{n-1,n-1} x_{n-1} + u_{n-1,n} x_n = c_{n-1} \quad \text{which gives } x_{n-1} = (c_{n-1} - u_{n-1,n} x_n) / u_{n-1,n-1}$$

Continuing this

way, we can

find x_i from:

$$x_i = (c_i - \sum_{j=i+1}^n u_{i,j} x_j) / u_{i,i}$$

Our algorithm in pseudocode is as follows:

```
procedure SerialGauss (A, b, x);
```

{ Solves the system $Ax = b$ by reduction to Upper triangular form and then uses back substitution. A is of size $n \times n$, b and x are n -vectors. }

```

1. for k := 1 to (n - 1) do
    1.1 Find pivot  $A[p,i] := \max \{ *A[j,k] * : k \# j \# n \}$ 
    1.2 Exchange the  $p$ -th row and  $k$ -th row of  $A$ 
    1.3 for i := (k+1) to n do
        Determine multiplier:  $M[i,k] := A[i,k]/A[i,i]$ ;

    1.4 Transform  $A, b$  :
        for j := (k+1) to n do
             $A[i,j] := A[i,j] - M[i,k]*A[k,j]$ ;
             $b[i] := b[i] - M[i,k]*b[k]$ 
        end for {j loop}
    end for {i loop}
end for {k loop}

2. Back substitution:

2.1  $x[n] := b[n]/A[n,n]$ ;

2.2 for i := (n-1) downto 1 do
     $x[i] := b[i]$ ;
    for j:= (i+1) to n do
         $x[i] := x[i] - A[i,j]x[j]$ ;
    end for
     $x[i] := x[i]/A[i,i]$ 
end for

end;
```

You must observe that the algorithm could fail if the pivot is zero. The pivot is either zero, or close to machine zero. In this case, the matrix A is singular and we abort the process. Also, the multipliers $M[i,k]$ need not be stored separately. They can be stored over $A[i,k]$ since this would be zero as a result of the transformation (1.4).

Analysis: This algorithm is labor extensive! For each value of i in the loop (1.3), the loop (1.4) executes $(n-i)$ times. Since (1.3) itself is done $(n-k)$ times, together the two nested loops are done

$(n-k)^2$ times. We can assume that the work inside the loop takes constant time. However, k itself loops from 1 to $n-1$. Hence the total time taken for transforming the system into upper triangular form alone is:

. By a similar analysis, we note that in the backsubstitution step 2, we have two nested loops. The inside j loop is done $(n-i)$ times while i loops $(n-1)$ times. The amount of time units for this is .

Thus the total time is of the order $O(n^3)$.

Parallelization: Assume that we have n processes. The most expensive part of the computation is the transformation. We will consider this first. A procedure to parallelize the backsubstitution will be considered later. The nested loops in steps (1.3), (1.4) are dependent upon the value of k . The aim of the k -loop is to find successive pivot rows. This will have to be done by a single process (say the mother). This process will also make row exchanges if necessary.

The inner loops (1.3) and (1.4) afford the greatest opportunity to parallelize. We could simply split the i -loop only. In that case every process will do its own inner j -loop. The total time in the j -loop is of order $O(n)$, and this is done in parallel. In view of the k -loop (1), this way of parallelization of the algorithm would result in a complexity of $O(n^2)$. This alone represents a speedup of n . However, as we proceed through the algorithm, the number of rows to be transformed reduces and there is a problem of load balancing. To avoid this, we could use the techniques of Chapter 5 for load balancing of nested loops.

Several barriers would be needed for synchronization. The first barrier call is to wait for the mother to find, and exchange if necessary, the pivot row. After this, processes are assigned to reduce the current system of equations. A second barrier call is needed here to wait for all the

processes to finish their work before the mother proceeds to find the next pivot row. A

pseudocode for a parallel version of Gaussian elimination is given below.

```
procedure ParallelGauss (A, b, x);

{ Solves the system  $Ax = b$  by reduction to Upper triangular
form and then uses back substitution. A is of size  $n \times n$ , b and x are
n-vectors. Assumes procs number of processors are available}

1. for k := 1 to (n - 1) do

    if id = 0 then
        1.1 Find pivot  $A[p,i] := \max \{ *A[j,k]* : k \neq j \neq n \}$ 
        1.2 Exchange the p-th row and k-th row of A
    end if
    WaitAtBarrier for Process 0 to finish
    i := k;
    i := i + id
    repeat
         $A[i,k] := A[i,k]/A[i,i]$ ;
        for j := (k+1) to n do
             $A[i,j] := A[i,j] - A[i,k]*A[k,j]$ ;
             $b[i] := b[i] - A[i,k]*b[k]$ 
        end for {j loop}
        i := i + procs
    until (i > n);
    WaitAtBarrier for all processes to finish

end for {k loop}

2. Back substitution:
if id = 0 then
    2.1  $x[n] := b[n]/A[n,n]$ ;

    2.2 for i := (n-1) downto 1 do
         $x[i] := b[i]$ ;
        for j:= (i+1) to n do
             $x[i] := x[i] - A[i,j]x[j]$ ;
        end for
         $x[i] := x[i]/A[i,i]$ 
    end for
end;
```

In the parallel version of Gaussian elimination, we have not parallelized the back substitution part. This part of the work is done serially by the mother process. The reason is this: After parallelization of the first part, the time complexity of that portion of the algorithm is $O(n^2)$. This is also the complexity of the backsubstitution part. Thus, parallelizing back substitution may not provide (which would merely change the time complexity of that part to $O(n)$) any more speedup.

This, however, assumes that we have only n processors. If we have more processors, we could parallelize both the i and j loops, just as we did in Chapter 5. We will divide the processors into groups so that the i is distributed among the groups fairly evenly. Each group of processors will divide the j iteration among them as evenly as possible. In this case, parallelization of backsubstitution is desirable.

Parallel back substitution: It would seem from our pseudocode, that back substitution is inherently sequential. First we compute x_n . After that, x_i is computed only after computing x_{i+1} . Nevertheless, we *can* parallelize this part of the algorithm also! Suppose that x_i is computed by process (i). The trick is to ensure that as soon as x_k , $k = n, n-1, \dots, i+1$ is computed, its value is

$$x_i = \{b_i - \sum_{j=i+1}^n a_{i,j} \cdot x_j\} / a_{i,i}$$

broadcast all the processes that need them. Since process (i) can immediately finish computation of x_i . Notice that this ensures computation of the above expression in *parallel*. The complexity of this operation is $O(n)$.

Implementation: How do we broadcast values to processors as soon as they are found? In

architectures that use pipelines, this is relatively simple. In our SIMD environment, we can achieve the same result by means of a shared array of records, *Solution* with n indices. Each record has two fields, one for the value of x_i , and the other a boolean to inform the other processes that the value has been posted. We assume that we have n processors. Process 0 computes x_n , and waits. Process (i) computes x_{n-i} . Each process scans the shared array to see which values have been posted and uses it in the formula given above. Once it has all the values to compute x_i (these are $x_{i+1}, x_{i+2}, \dots, x_n$), it completes its computation, updates *Solution*[i] appropriately. Notice that it cannot post its own value unless it sees that process (i+1) has posted its value. *It then dies*. There is no further need for the child processes.

```

type
  range = 1..n;
  pipe = array [range] of record
      value : real;
      posted : boolean
  end;

var Solution : pipe;
    PostedSet : set of range;

```

This array is initialized with *posted* = *false*. . . Our pseudocode for parallel back substitution is as follows:

```

procedure BackSubstitution (id, A, Solution, x);

  if id = 0 then      {Mother computes x[n]}
    Solution[n-id].Value := b[n]/A[n,n];
    Solution[n-id].posted := True;
    Wait;

  else {Child with id = i, computes x[n-i]}
    i := n - id;
    PostedSet = [i+1,..n];
    Solution[i].value := b[i];

```

```

repeat
  for j := (i + 1) to n do
    if j in PostedSet then
      solution[i].value := solution[i].value + A[i,j]*x[j];
      PostedSet:= PostedSet - [j];
    end if
  until (PostedSet = []);
  Solution[i].value := solution[i].value/A[i,i];
  Solution[i].Posted := True
end if-else
Kill process;
end;

```

At this point all the solutions have been calculated. We have assumed only n processes. With even more processes available, one would have to be more ingenious.