

CHAPTER 5. LOOP-SPLITTING - NESTED LOOPS.

In the last chapter, we considered the construction of a barrier. The barrier is useful to halt the execution of a process until a specified number of additional processes have completed their work as well. We also saw multiple uses of the same barrier. In this chapter, we revisit loop-splitting in nested loops. In particular, we consider load balancing when the number of iterations is fewer than the number of processes.

5.1 More processes than loops

In earlier chapters, we have seen that loop-splitting while quite easy to implement, often causes load unbalancing. In particular, if n is the number of times a loop is iterated, and p is the number of processes created, unless p divides n evenly, some processes would do at least an additional iteration than the others. On the other hand, if $n \ll p$, that is n is much smaller than p , then some processes will be idle, $(p - n)$ to be exact. This is the other side of load unbalancing. It seems inefficient to use more processes than needed. When the loops are nested, the total number of iterations involved may justify using additional processes. We illustrate this by considering multiplication of a matrix by a vector.

Assume that A is an $n \times n$ matrix, x a n -vector. Then $y = Ax$ is also a n -vector. The serial version is a nested loop:

```
for (i=0; i<n ; i++)
{
    y [i] = 0;
    for (j=0; j<n ; j++)
        y[i] = y[i] + A[i, j] * x[j]
}
```

Parallel Version 1. Our first parallel version simply splits the outer loop:

```

id := process_fork (p);

for (i=id; i<n ; i=i+p)
{
    y [i] = 0;
    for (j=0; j<n ; j++)
        y[i] = y[i] + A[i, j] * x[j]
}

```

Suppose that $n = 10$ and $p = 16$ so that we have many more processes than the loop iterations.

Evidently, neglecting overhead, with this parallel version, the speedup is at most 10. If we hope to increase the speedup at all, we need to parallelize the inner j -loop as well.

Parallel Version 2: In the parallel version given above, each process does exactly one value of i . An interesting loop variation is the one in which several processes do the same value of i , and share in doing the inner loop. Our plan is as follows. We divide the processors into four groups. The number of groups is arbitrary for now, and will be discussed further when we consider speed up obtained by this arrangement.

| <i>Processes used</i> | <i>Values of i assigned</i> | <i>Values of j assigned</i> |
|-----------------------|-----------------------------|---|
| 0, 1, 2, 3 | 0, 4, 8 | Process 0: j = 0,4,8 Process 1 : j = 1,5,9 Process 2 : j = 2,6 Process 3 : j = 3,7 |
| 4, 5, 6, 7 | 1, 5, 9 | Process 4: j = 0,4,8 Process 5 : j = 1,5,9 Process 6 : j = 2,6 Process 7 : j = 3,7 |
| 8, 9, 10, 11 | 2, 6 | Process 8: j = 0,4,8 Process 9 : j = 1,5,9 |

| | | |
|----------------|------|-----------------------|
| | | Process 10 : j = 2,6 |
| | | Process 11 : j = 3,7 |
| 12, 13, 14, 15 | 3, 7 | Process 12: j = 0,4,8 |
| | | Process 13: j = 1,5,9 |
| | | Process 14 : j = 2,6 |
| | | Process 15 : j =3,7 |

Of course, this scheme is for $n = 10$ and $p = 16$. Similar schemes can be set up for other cases where $n \ll p$. You should note the apportionment of i 's among the processes: Two groups of three each and two groups of two each. The j -s are grouped the same way within each group.

Four groups of processors are used (0 - 3), (4 - 7), (8 - 11) and (12 - 15), each for a single value of i . To avoid race conditions, we would need four barriers, one for each group. For example, if b is an array of four barriers, the first group of process (0 - 3) will use barrier $b[0]$; the second group ((4 - 7) will use $b[1]$, and so on. Each barrier will stop the processors of its own group, which is three. Therefore, the barriers must be initialized elsewhere in the program:

```
for (counter = 0; counter <3; counter++)
    b[counter]->Barrierinit(4);
```

where $counter$ is a variable of type integer. The processes of a given group will all be stopped by the same barrier. We ensure this by using the *div* function:

```
b[id/4]->WaitAtBarrier ();
```

The first group of processes, $id = 0$ through 3, uses $b[0]$. The second group, $id = 4$ through 7, use $b[1]$, and so on.

The i loop is split among the four groups in the same way and they jump by $p/4$ or 4 each time. The inner j loop is also split in a similar way. In what follows, we the variables A , x , and y are shared. The variable *tempsum* is a shared array of p indices, used by the

processes as accumulators.

```
for (i=id/k; i<n; i=i+g)           // g groups, k processes in a group
{
  y[i]=0;
  for (j=id%k; j<n; j=j+k)
    tempsum[id] = tempsum[id] + A[i,j]*x[j]
b[id/k] -> WaitAtBarrier()
if (id % k ==0)
  for (counter=0;counter<k;counter++)
    y[i] += tempsum[id+counter]
b[id/k] -> WaitAtBarrier()
}
```

How fast is this version? The 16 processes are divided into 4 groups with 4 processes per group. Neglecting overhead, the serial version takes n^2t time units, where t is the time to compute $y[i]$ inside the j -loop. The 16 processes were divided into four groups, with four processes per group. The four groups must do all the i -loops, so that there are $10/4$ or 3 i -iterations. On the other hand, the four processes in a group must do all the j -values, so that there are again $10/4$ or 3 j -iterations. Therefore, the speedup for this set up is $100t/9t = 11.1$. Although this is an improvement, it is only marginal. Can we increase the speedup even more?

As we did in Chapter 2, we can compute a formula for the speed up and this formula suggests how many processes one should choose.

Theorem. Let the number of outer and inner iterations be n and m respectively, p be the number of processes, and assume that all the processes are divided into g groups. Suppose that the n outer loop iterations are divided among the g groups of processes. Assume further, that the processes in any group divide m inner loop iterations among themselves. If all divisions are done as evenly as possible, then the maximum speedup σ is given by:

$$\sigma = nm / \{ (\frac{m}{p/g} + \frac{n}{g}) \}$$

Proof: Since the g groups of processes must do all the outer loop iterations, the maximum number of outer loop iterations is $\frac{n}{g}$. Since all divisions are as even as possible, the number of processes in a group is at most $\lceil \frac{p}{g} \rceil$, and these must do the m inner loops evenly. Therefore, the maximum number of inner loop iterations is $\frac{m}{\lceil \frac{p}{g} \rceil}$. Since the total number of iterations done serially is nm , the result follows.

In our case $n = m$, and in the special case above, $g = 4$. The speedup then is $\sigma = 100 / \{ \frac{100}{\lceil \frac{100}{4} \rceil} + \frac{100}{4} \} = 100 / (3 + 25) = 11.1$. How can we increase the speedup? The answer depends on the choice of g . By carefully choosing g , one can make the speedup equal to p , the number of processes. Remember however, that the speedup is limited by the number of processors available.

We consider the special case with $n = m$ so that the number of serial loop iterations is n^2 .

We can simplify the expression for σ by choosing p and g carefully. Suppose that $g = (p, n)$, that is g is the greatest common divisor of p and n . Then $g, p/g$ and $g, n/g$. It is known that

integers x, y such that $g = px + ny$. Therefore, $gn = pnx + n^2y$. Thus if $p \mid n^2$, then $p \mid gn$. In the expression for σ , we can drop the ceiling functions and get $\sigma = n^2 / \{n/(p/g) (n/g)\}$, that is $\sigma = n^2 / (gn/p)(n/g) = n^2 p / n^2 = p$. For example, if $n = 10$, choose $p = 20$. Now choose $g = (20, 10) = 10$. Then you can easily verify that $\sigma = 20$. However, if the number of processors $\kappa < p$, the speedup can never be more than p .

Parallel Version 3. In this version, we show how self-scheduling can be used for the two loops. The idea behind this version is that in computing $y[i]$,

$$y[i] = \sum_{j=0}^n A(i, j) * x[j], \quad i = 0, 1, \dots, n-1$$

there are in all $n * n = n^2$ multiplications done. The actual self-scheduling is accomplished by letting any free process choose an available i, j and do the multiplication $A[i, j] * x[j]$. When all the n multiplications for a given i have been done, the results are collected to compute $y[i]$ using n critical regions (since the $y[i]$ will be shared) one for each i .

In our examples involving arrays, we accomplished self-scheduling by controlling the last index of the array that was processed. The interesting question here is, "How does one control indices of two arrays, one each for i and j ?" We do this by actually controlling n^2 ! For each $k, 0 \leq k \leq n^2 - 1$, we can find a unique ordered pair (i, j) , that is we construct the function:

$$f[0, n^2 - 1] \rightarrow \{(i, j) : 1 \leq i, j \leq n\}$$

This function is both 1-1 and onto, that is each k defines a unique value of (i, j) , and

conversely. Thus, given k , we can then find i and j and then use i for the row and j for the column of the two-dimensional array A .. The function f is defined as follows:

$$f(k) = (k/n, k\%n).$$

For example, if $k = 76$, $n = 10$, then $i = 7$, and $j = 6$. It is important to note that 76 does *not* refer to the element in position (7,6)! Actually, 76 refers to that element which when the elements of A are counted row-wise, will be the 76-th element. This will be the element in row 7 and column 6.

With the n critical regions for the y -s, and one for k , we need in all $(n + 1)$ spinlocks. No barriers are needed. The arrays A , x and y will be shared. The complete program code is given below.

```
// Selfscheduling technique on nested loops
// Finds the product of the n x n matrix A and the n-vector x.
//Result is stored in the n-vector y

#include <iostream.h>
#include <math.h>
#include "utility.h"
#include "mbarrier.h"

const int size = 5;
const int nproc = 10;
int a[size, size], x[size];
int* ind;
int* y;
void main()
{
    int i, j, temp;
    int id;
    int tempindex;
    int gate[size], indexsem;
    int shareid[2];
    y = (int*)shareint(size*sizeof(int), shareid[0]);
    ind=(int*)shareint(sizeof(int), shareid[1]);

    //initialize array data using cin ?
    for(i=0; i<size; i++)
        for (j=0; j<size; j++)
            a[i, j]=1; //easy initialization?
```

```

        for (j=0; j<size; j++)
        {   x[j]=1;
            y[j]=0;
        }

spin_lock_init(indexsem,0);    //semaphore for determining index

for(i=0; i<size; i++)
    spin_lock_init(gate[i],0); //semaphore for each row.

*ind=size*size -1;    //initialize *ind

id=process_fork(nproc);

while (*ind >=0)
{   spin_lock(indexsem);

    tempindex = *ind;

    *ind--;
    spin_unlock(indexsem);
    i=tempindex/size;

    j=tempindex%size;

    temp=a[i,j]*x[j];
    cout<<i<<' '<<j<<' '<<*ind<<' '<<tempindex<<endl;
    spin_lock(gate[i]);
    y[i] +=temp;
    spin_unlock(gate[i]);    // using just a single semaphore
                             // makes is inherently sequential

}

process_join(nproc,id);
for (i=0; i<2; i++)          // Release shared memory
    clean_up_shared(shareid[i]);

for (i=0; i<size; i++)
    clean_up_sem(gate[i]);    // Release Semaphores
clean_up_sem(indexsem);

for (i=0; i<size; i++)
    cout<<y[i]<<endl;        //output y

}

```

Remarks

1. The program uses self-scheduling to compute the next index available to each process.

However, the amount of work done inside the protected region (the region protected by

lockid [0]) is quite small. In the chapter on spinlocks, we learned that it is not particularly efficient to use spinlocks this way. This could actually cause a bottle neck! If one process begins to idle inside, other processes would be forced to spin outside, waiting to get a value of index that they can process.

2. The use of self-scheduling is for illustrative purposes only. It may, in fact, be more efficient to use loop-splitting to distribute the index values among all the processes:

```

for (i=id; i< n*n; i+=procs)
{
    row = i / n;
    col = i % n;
    spin_lock (lockid [row]);
    y[row] = y[row] + A[row, col] * x[col];
    spin_unlock (lockid [row]);
};

```

This will remove the bottleneck referred to above.

3. Both in self-scheduling in the main program, and in the loop-splitting shown above, the values of *row* and *col* are computed by each process for its appropriate values of *the* index *i*. Although this computation is very minimal in times of time spent, one could consider computing ahead the values of row and col for each value of *i*. This is done by setting up two arrays, one for row values and the other for column values. The values stored in the two arrays are computed as shown above:

$$\text{row [i]} = i / n, \quad \text{col [i]} = i \% n.$$

4. What is the speed of the self-scheduling version? If p is the number of processes used, and the n^2 indices are apportioned among the processes evenly, the number of iterations done in parallel is n^2/p . Therefore the maximum speedup = $n^2/(pn^2/p)$, which is of course no more than p .

5. The loop splitting idea described in Parallel Version 2 (group scheduling technique) can be extended to multiple levels. For example, a three level nesting loops such as

```
for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    for (k=0;k<n;k++)
      job(i,j,k)
```

can be parallelized by scheduling processes to work at all three levels. Assuming you have many processes ($>2n$), the processes can be divided into *clusters*. Each *cluster* can be divided into *groups*, with each *group* containing a few processes. The idea would be to do loopsplitting of i among *clusters*, loopsplitting j among *groups*, loopsplitting k amongst processes in that *group*. This is left as an exercise. A practical implementation is to use such a technique to do matrix multiplication: $A \times B = C$ (A, B, C are $n \times n$ matrices).