Parallel Multiplication: A Case Study in Parallel Programming

C K Yuen and M D Feng
Department of Information Systems and Computer Science
National University of Singapore, Kent Ridge, Singapore 0511
email: `iscyck@nusvm.nus.sg` and `fengmd@iscs.nus.sg`

Abstract

Six versions of a parallel program for multiplication are presented, and compared in terms of their efficiencies. They illustrate the bottom up and top down methods of program structure design, and the use of tuplespace and speculative processing in parallel programming. The ideas are also applicable to general `AND`/`OR` parallel problems.

1 Introduction

We wish to consider the problem of multiplying a number of factors, each of which takes some time to compute, and it is desirable to have them evaluated in separate tasks. The multiplication program would thus have to generate the tasks, receive their results and combine these. This reflects the *divide and conquer* approach for general problem solving, the fundamental technique in algorithm design [1], and in exploitation of parallelism in a multiprocessor system [6].

In our study, we shall use the language BaLinda Pascal, which derives from our research project on BIDDLE architecture [11] and BaLinda Lisp language [12]. BaLinda Pascal provides the `EXEC` construct for programmers to indicate the parallelism, and Linda [4] constructs for interaction among parallel modules. However, most of the examples can be readily recoded into other parallel languages.

To show our starting example, supposing there are `N+1` factors, and each could be computed by the calling function `Factor(0)`, ..., `Factor(N)` respectively:

Parallel Multiplication Version 1 (with $O(N)$ tuple time)

```
PROCEDURE PMultiply (I: INTEGER);
   VAR X, Product: REAL;
BEGIN
   X := Factor (I);
   IN ('Product' ? Product);
   OUT ('Product', Product * X)
END;
```

```
...
OUT ('Product', 1);
FOR I := 0 TO N DO
   EXEC PMultiply (I) ENDOFTASK;
SYNCHRONIZE (N + 1);
IN ('Product' ? Product);
...
```

Here, the `EXEC` keyword is used to identify independently executable parts of a program. `ENDOFTASK` ends a parallel task, `SYNCHRONIZE` allows a task to wait for the termination of its *most recent* subtask. `EXEC` is analogous to the `fork` in UNIX C [2], `ENDOFTASK` to `exit`, and `SYNCHRONIZE` to `wait`, but with the following differences:

1. `EXEC` does not cause a second copy of the environment to be created for the new task on a shared memory system. The two tasks share the same environment where possible. In a distributed system, the part of the environment lexically accessible to the new task may be duplicated, except for those items declared as `SHARING`.

2. `SYNCHRONIZE` waits for the most recent subtask to terminate. If a parameter n is provided, then it waits for the n most recent tasks to terminate.

3. `EXEC` and `SYNCHRONIZE` do not return any task identity or other system information. This is the consequence of the previous point.

The `OUT` and `IN` are Linda commands which operate on a logically shared data space (called *tuplespace*) containing multi-field records (called *tuples*). A task puts a tuple into the tuplespace using an `OUT` command:

$$\texttt{OUT}\ (\langle \mathrm{exp}_1 \rangle\ \langle \mathrm{exp}_2 \rangle\ \ldots\ \langle \mathrm{exp}_n \rangle)$$

where each expression defines a value of any type, whether numerical, logical, character, string or even array/list. Tuples are not accessed by name or address, but by content, using the `IN` command:

$$\texttt{IN}\ (\langle \mathrm{exp}_1 \rangle\ \langle \mathrm{exp}_2 \rangle\ \ldots\ \langle \mathrm{exp}_m \rangle\ \texttt{?}\ \langle \mathrm{var}_1 \rangle\ \langle \mathrm{var}_2 \rangle\ \ldots\ \langle \mathrm{var}_{n-m} \rangle)$$

This will retrieve from the tuplespace an n-field tuple whose first m fields match the result of the m expressions in the `IN`, and then store the values of the last $n - m$ fields into the $n - m$ variables specified in the `IN`. If no matching tuple can be found, the task executing the `IN` is suspended until another task `OUT`s the required tuple.[1]

In the above program, the main task first initializes the product tuple, then spawns `N+1` parallel tasks, each of which computes one factor and multiplies the result with the partial product in the product tuple. The main task waits for all the `N+1` tasks to end by a `SYNCHRONIZE` before it fetches the final product in the product tuple.

[1] In Linda, besides `OUT` and `IN`, there is an `RD` command which behaves the same as `IN` except `IN` causes the tuple to be removed from the tuplespace, while `RD` leaves it for others to access. We will not use the `RD` command in all the examples here, but it is provided in BaLinda Pascal.

2 Optimizing Tuple Operations

The first version is simple, but there is a problem that the time taken to obtain the product tuple and change it increases with `N` because the product tuple could be `IN`ed by only one task at any time. Thus, the tuple interaction time is $O(N)$, which would become the bottleneck if `N` is very large. We have therefore coded the following method in which the tuple operations take in $O(\log N)$ time:

Parallel Multiplication Version 2 (with $O(\log N)$ tuple time)

```
PROCEDURE PMultiply (I, Max: INTEGER);
   VAR X, Y: REAL;
       EXIT: BOOLEAN;
BEGIN
   X := Factor (I);
   Exit := FALSE;
   REPEAT IF ODD (I)
          THEN BEGIN
                  OUT (Max, X);
                  Exit := TRUE
               END
          ELSE IF (Max > 0)
               THEN BEGIN
                       IF (I <> Max)
                       THEN BEGIN
                               IN (Max ? Y);
                               X := X * Y
                            END;
                       I := I DIV 2;
                       Max := Max DIV 2
                    END
               ELSE BEGIN
                       OUT ('Product', X);
                       Exit := TRUE
                    END
   UNTIL Exit
END;

...
FOR I := 0 TO N DO
   EXEC PMultiply (I, N) ENDOFTASK;
SYNCHRONIZE (N + 1);
IN ('Product' ? Product);
...
```

The basic idea is that pairs of tasks would synchronize with each other first, then groups of 4, 8, etc. That is, each odd-number task `OUT`s a tuple containing the computed factor, and even-number task `IN`s a tuple and multiplies with its own factor. Afterwards, the former task simply exits, while the latter halves its index (i.e. `I`) and proceeds to the next level. If the number of tasks is even, then each task has a partner; when the number of tasks is odd, the task with the maximum index (i.e. `Max`), which is an even number, has no partner, and it skips this level by halving its index and immediately proceeding to the next level. The number of tasks (i.e. `MAX+1`) entering the higher level always decreases by half. This process continues until there is only one task left (i.e. `MAX` = 0), at which point it issues the product tuple containing the final product. Thus, multiplying n factors only requires $\lceil \log n \rceil$ levels of tuple interaction. The process is graphically described in Figure 1 for `N` = 6 (7 factors, or 7 tasks).

```
Task     |     0         1         2         3         4         5         6
---------|--------------------------------------------------------------------
(I, Max) |  (0, 6)    (1, 6)    (2, 6)    (3, 6)    (4, 6)    (5, 6)    (6, 6)
         |     |         |         |         |         |         |         |
Level 1  |    IN <----- OUT       IN <----- OUT       IN <----- OUT        |
         |     |         |         |         |         |         |         |
(I, Max) |  (0, 3)   ENDOFTASK  (1, 3)   ENDOFTASK  (2, 3)   ENDOFTASK  (3, 3)
         |     |                   |                   |                   |
Level 2  |    IN <--------------- OUT                 IN <--------------- OUT
         |     |                   |                   |                   |
(I, Max) |  (0, 1)             ENDOFTASK           (1, 1)             ENDOFTASK
         |     |                                       |
Level 3  |    IN <------------------------------------ OUT
         |     |                                       |
(I, Max) |  (0, 0)                                 ENDOFTASK
         |     |
         |    OUT
         |     |
         | ENDOFTASK
```

Figure 1: Parallel multiplication with $O(\log N)$ tuple time

3 Top Down Design

The second program evolved naturally from the first, but let us consider the way the structure was arrived at: We first generate the tasks that compute the factors, then we try to re-integrate their results. This is the *bottom up* approach. Obtaining results from the tasks requires tuples to be used, but this is an expensive method for information sharing that should be minimized. Although the tuple interaction time is reduced to $O(\log N)$, the time required to generate and synchronize `N+1` tasks is still $O(N)$. This may be of little consequence when the computation of a factor is time consuming and when `N` is small, but massively parallel applications are bound to suffer.

Let us re-consider the problem in the *top down* fashion: Given factors 0 to N to compute and multiply, we can have two tasks to produce two smaller products, then combine them. Each task can in turn generate two tasks for even smaller products, and so on. The following program results:

Parallel Multiplication Version 3 (with top down design but without tuple)

```
FUNCTION PMultiply (Min, Max: INTEGER) : REAL;
   VAR Middle: INTEGER;
       X, Y: REAL;
BEGIN
   IF (Min = Max)
   THEN PMultiply := Factor (Min)
   ELSE BEGIN
           Middle := (Min + Max) DIV 2;
           EXEC SHARING (X) X := PMultiply (Min, Middle) ENDOFTASK;
           Y := PMultiply (Middle + 1, Max);
           SYNCHRONIZE;
           PMultiply := X * Y
        END
END;

...
Product := PMultiply (0, N);
...
```

Note that X is declared as a SHARING variable to prevent its duplication because the main task must receive the value produced by the subtask. If X is duplicated, then the new value produced would only appear in the new task.

This program may not look as clever as the previous one, but actually works better because of its elimination of all the tuple overheads. First, only N tasks will be spawned (not including the main task) in total which is one less than the task number in the previous two versions. Next, it allows N+1 tasks to be started in $O(\log N)$ time. This is actually the best that can be expected of the EXEC construct because each EXEC splits a thread of computation into two. Third, the N+1 tasks could also be synchronized in $O(\log N)$ time. We could expect that the task creation and synchronization cost could be distributed more evenly than the previous two versions where the main task takes all the responsibilities. It has clearly shown the importance of considering problems top down, with the possibility of making a fresh start out of an existing situation rather than merely evolving in a bottom up fashion.

The top down method has been adopted in structured programming [5] and software system design [9]. It is argued in [5][7][10] that top down development is a superior approach because it results in the creation of more readable and more reliable programs than those implemented using bottom up techniques. The parallel multiplication program here shows that the top down design also makes the parallel programs more efficient.

4 Speculative Processing

The above parallel multiplication process is also applicable to summation, `AND`, `OR`, and other commutative and associative operations. However, an additional elaboration is relevant: If any factor in a product is 0, then the result must be 0, and it is unnecessary to compute the other factors. Similarly, in an `AND` any input being `FALSE` makes the result `FALSE`, and in `OR` any `TRUE` input makes output `TRUE`. How can this process of simplification be incorporated into the program?

The following three programs use a technique called *speculative parallelism*. Speculative parallelism is the initiation of parallel computation on the basis of speculation about the usefulness of its result [3]. In other words, a task is spawned in the hope that its result will later be of use. The speculative computation could be classified into three types based on the way in which resources are used for speculation [8]: multiple-approach, order-based and precomputing. The parallel multiplication fits into the first category in which the speculation is in pursuing multiple approaches (i.e. computing the factors) simultaneously while not all approaches may be needed at the end. It may reach a time in the future to abort irrelevant computation (i.e. whenever one factor is founded as 0).

This speculation is expressed in BaLinda Pascal using the `IF` construct:

```
IF ⟨boolean⟩
THEN ⟨then-part⟩
ELSE ⟨else-part⟩
```

It may be useful to execute the ⟨boolean⟩ module in parallel with the other two modules (i.e. ⟨then-part⟩ and ⟨else-part⟩) so that, when the ⟨boolean⟩'s `TRUE/FALSE` value is known, the wanted result is already available or partially computed, and no or very little waiting is necessary. To cause this, we place the `EXEC` in front of `THEN` or `ELSE`, or both, with a corresponding `ENDOFTASK` after the `THEN/ELSE` module, together with a `SYNCHRONIZE` after the ⟨boolean⟩ module.[2] For example, if we intend to execute ⟨then-part⟩ speculatively, then the construct is:

```
IF ⟨boolean⟩ SYNCHRONIZE
EXEC THEN ⟨then-part⟩
ELSE ⟨else-part⟩
```

However, the `THEN` and `ELSE` modules should be executed at lower priority because we do not know which will be selected. If the machine is already heavily loaded, we do not add to the competition for resources because a lower priority task does not get selected for execution, but if the machine has idle capacity, then even lower priority tasks can be executed without affecting others. If the `THEN` and `ELSE` modules are still being executed when the ⟨boolean⟩ result is returned, the selected module is confirmed by raising its execution priority, while the unwanted module is purged. We also impose the rule that the side effects (i.e. variable assignments and tuple operations) of each speculative module are retained in its own data space until confirmation. Thus, a deleted task does not produce visible side effects.

[2]We do not have `EXEC` after the `THEN` or the `ELSE` because this might give the incorrect impression that parallel tasking occurs only when the ⟨boolean⟩ is `TRUE` or `FALSE`.

We now consider the use of this idea in the parallel multiplication program. It may be wasteful to compute all of the factors at high priority, because if one turns out to be zero, then the rest are not needed. Hence, we compute one at full priority, and the rest at lower priority, but if the first one turns out to be non-zero, then the next task's priority is raised. If the second factor also turns out to be non-zero, then the priority of the third factor rises. And so on.

The `PMultiply(N)` function call spawns `N` tasks by recursively calling itself, but only the task computing the first factor (i.e. `Factor(0)`) is at normal priority as the others are enclosed in the parallel ⟨then-part⟩ of a conditional statement. `PMultiply(I)` returns 0 if `PMultiply(I-1)` has returned 0, or if `Factor(I)` is 0; otherwise, it return a non-zero value. If the latter occurs before `PMultiply(I+1)` completes, then `PMultiply(I+1)`'s execution priority rises. If `PMultiply(I)` returns zero, then the computation of factors `(I+1)` and higher is abandoned.[3]

It is of course possible that, because the machine is lightly loaded, the higher factors have already been computed even though they are not needed, but if there is competition for the processor, then the higher factors do not get computed until they are confirmed by the previous tasks, computing at higher priority, returning `TRUE`. The waste would therefore not occur.

Parallel Multiplication Version 4 (with speculative processing and $O(N)$ tuple time)

```
FUNCTION PMultiply (I: INTEGER) : REAL;
   VAR X, Y: REAL;
BEGIN
   IF (I = 0)
   THEN BEGIN
           X := Factor (0);
           OUT ('Product', X);
           PMultiply := X
        END
   ELSE IF (PMultiply (I - 1) <> 0) SYNCHRONIZE
        EXEC THEN BEGIN
                     X := Factor (I);
                     IN ('Product' ? Y);
                     OUT ('Product', X * Y);
                     PMultiply := X
                     ENDOFTASK
                  END
        ELSE PMultiply := 0
END;

...
IF (PMultiply (N) <> 0)
THEN IN ('Product' ? Product)
ELSE Product := 0;
...
```

[3]It is, however, not possible for the discovery of factor `I` being zero to cause the abandonment of the earlier tasks.

We have left the product tuple access bottleneck, since only the highest priority task may successfully retrieve the product tuple, while the other tasks would be blocked in their respective `IN`s due to the confined side effects of speculative tasks. However, the total tuple interaction would take $O(N)$ time.

The following program avoids the use of the product tuple, but is somewhat harder to understand: The trick lies in using the function `NotZero` to call `PMultiply(I-1)` and retain the returned value in `X`, so that it can be used to compute the product, as well as to check whether it is 0.

Parallel Multiplication Version 5 (with speculative processing but without tuple)

```
FUNCTION PMultiply (I: INTEGER) : REAL;
   VAR X, Y: REAL;
   FUNCTION NotZero: BOOLEAN;
   BEGIN
      X := PMultiply (I - 1);
      NotZero := X <> 0.0
   END;
BEGIN
   IF (I = 0)
   THEN PMultiply := Factor (0)
   ELSE BEGIN
           IF NotZero SYNCHRONIZE
           EXEC THEN Y := Factor (I) ENDOFTASK
           ELSE Y := 0;
           PMultiply := X * Y
        END
END;

...
Product := PMultiply (N);
...
```

Although the above program gets rid of any tuple operations, `N` speculative tasks are still recursively generated by the main task in $O(N)$ time (assuming there are enough processors to handle the lower priority tasks). By the top down method, it is possible to reduce the task spawning procedure into $O(\log N)$ time:

Parallel Multiplication Version 6 (with speculative processing and top down design but without tuple)

```
FUNCTION PMultiply (Min, Max: INTEGER) : REAL;
   VAR X, Y: REAL;
       Middle: INTEGER;
   FUNCTION NotZero: BOOLEAN;
   BEGIN
```

```
      X := PMultiply (Min, Middle);
      NotZero := X <> 0.0
   END;
BEGIN
   IF (Min = Max)
   THEN PMultiply := Factor (Min)
   ELSE BEGIN
           Middle := (Min + Max) DIV 2;
           IF NotZero SYNCHRONIZE
           EXEC THEN Y := PMultiply (Middle + 1, Max) ENDOFTASK
           ELSE Y := 0;
           PMultiply := X * Y
        END
END;

...
Product := PMultiply (0, N);
...
```

However, note that because of the nesting of parallel `IF ... THEN ...`, with `IF` on the left node and `THEN` on the right node, the priorities of factors on right branches get progressively lower as we go down the tree, and that the discovery of a 0 factor causes the abandonment of computation on right nodes only.

The parallel multiplication program can also be generalized to handle the computation of `AND`-parallel and `OR`-parallel expressions. In the former, if any term computes to `FALSE`, then the remaining computation may be abandoned. Producing the program only requires replacing multiplication by Boolean operation, and check `TRUE/FALSE` result instead checking for zero. Below is the parallel `AND` program, in two versions:

Parallel `AND` Version 1 (with speculative processing and $O(N)$ tuple time)

```
FUNCTION ParaAND (I: INTEGER) : BOOLEAN;
   VAR X, Y: BOOLEAN;
BEGIN
   IF (I = 0)
   THEN BEGIN
           X := Factor (0);
           OUT ('Result', X);
           ParaAND := X
        END
   ELSE IF ParaAND (I - 1) SYNCHRONIZE
        EXEC THEN BEGIN
                     X := Factor (I);
                     IN ('Result' ? Y);
                     OUT ('Result', X AND Y);
```

```
                     ParaAND := X
                     ENDOFTASK
                  END
        ELSE ParaAND := FALSE
END;

...
Result := ParaAND (N);
...
```

Parallel `AND` **Version 2** (with speculative processing but without tuple)

```
FUNCTION ParaAND (I: INTEGER) : BOOLEAN;
   VAR X, Y: BOOLEAN;
   FUNCTION NotFALSE: BOOLEAN;
   BEGIN
      X := ParaAND (I - 1);
      NotFALSE := X
   END;
BEGIN
   IF (I = 0)
   THEN ParaAND := Factor (0)
   ELSE BEGIN
           IF NotFALSE SYNCHRONIZE
           EXEC THEN Y := Factor (I) ENDOFTASK
           ELSE Y := FALSE;
           ParaAND := X AND Y
        END
END;

...
Result := ParaAND (N);
...
```

The top down version and the parallel `OR` programs could be developed similarly.

5 Conclusions

We have shown six versions of a parallel multiplication program, evolving from the simplest case to the sophisticated one reducing tuplespace operations and task initiation time. They illustrate the merit of the top down technique in the problem solving and the advantage of speculative processing to cater for different processing requirement and produce efficient parallelism. These ideas are applicable to large problems usually solved by the divide and conquer technique.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1975.

[2] M. J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[3] F. W. Burton. Speculative computation, parallelism, and functional programming. *IEEE Transactions on Computers*, c-34(12):1190–1193, 1985.

[4] N. Carriero and D. Gelernter. Linda in context. *Communication of the ACM*, 32(4):444–458, 1989.

[5] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[6] H. T. Kung. Computational models for parallel computers. In R. Elliott and C. A. R. Hoare, editors, *Scientific Applications of Multiprocessors*, pages 1–15. Prentice-Hall, 1989.

[7] P. Naur. An experiment on program development. *BIT*, 12:347–365, 1972.

[8] R. Osborne. Speculative computation in Multilisp. In T. Ito and R. Halstead Jr., editors, *Parallel Lisp: Languages and Systems*, pages 103–137. Springer-Verlag, 1990. LNCS 441.

[9] I. Sommerville. *Software Engineering*, pages 181–182. Addison-Wesley Publishing Company, third edition, 1989.

[10] N. Wirth. Program development by stepwise refinement. *Communication of the ACM*, 14(4):221–227, 1971.

[11] J. J. Yee and C. K. Yuen. BIDDLE: a dataflow architecture for Lisp. In *Proceedings of the Twenty-Fifth Annual Hawaii International Conference on System Sciences*, pages 611–618, Kauai, Hawaii, January 1992.

[12] C. K. Yuen, M. D. Feng, W. F. Wong, and J. J. Yee. *Parallel Lisp Systems: A Study of Languages and Architectures*. Chapman and Hall, 1993.