ソフトウェア基礎論 31－4
(1989. 9. 20)

Evaluation Partial Order and Synchronization Mechanisms in Parallel Functional Programs

Masato Takeichi

Educational Computer Centre and
Faculty of Engineering
University of Tokyo
Bunkyo-ku, Tokyo 113, Japan

ABSTRACT

We show how synchronization affects the evaluation partial order in parallel functional programs and examine in detail the synchronization primitive proposed by Hughes. Despite the fact that it may cause deadlock, it is proved useful even on a single processor implementation because it makes programs use less space. Moreover on parallel processors it increases parallelism and makes the total evaluation time shorter. Instead of trying to prove programs free of deadlock, we give a practical method to construct deadlock-free programs. At present we do this informally, but our method based on strictness analysis and annotation of parameter mechanism seems promising. We also propose a translation rule for a higher level notation into parallel code which will run synchronously. This relies on an implicit synchronization mechanism inherent in strict functions. Execution profiles on an experimental system support our idea.

並行関数プログラムにおける評価順序と同期機構について

武市　正人

東京大学　教育用計算機センター・工学部

本論文では並行関数プログラムの評価順序に同期機能が及ぼす効果を示し、Hughesが提案している同期機構を詳しく調べる。Hughesの同期機構はデッドロックを生じさせることがあるとはいえ、少ない領域しか必要としないプログラムを作ることができるので、単一のプロセッサ上でも有用であるとされている。さらに、並列プロセッサ上では並列度を増し、全計算時間を短くすることができる。本稿ではプログラムがデッドロックを引き起こさないことを証明するのではなく、デッドロックの生じないプログラムを構成する方法を与える。ここでは形式的に扱ってはいないが、strictness解析と引数機構に基づくこの方法は有望であると考えられる。また、ある高水準の表記法を同期的に働く並行プログラムに変換する規則も提示している。これはstrictな関数に内在する暗黙の同期機構によるものである。これらの考え方の実験システムによる例証も示している。

Evaluation Partial Order and Synchronization Mechanisms in Parallel Functional Programs

Masato Takeichi

Educational Computer Centre and
Faculty of Engineering
University of Tokyo
Bunkyo-ku, Tokyo 113, Japan

1. Introduction

It has been suggested that functional parallel programming is much easier than writing imperative parallel programs. This belief is founded on the fact that any functional program has an important property called *referential transparency*; the meaning of a program (expression) depends only on the values of its constituent expressions, and these subexpressions may be freely replaced by others with the same value. The values of such subexpressions depend only on the context and irrelevant to any procedures for obtaining them or the order of replacing expressions with their values. Hence functional programmers feel no concern about whether a particular expression is evaluated before others or not. This holds both for sequential and for parallel programs.

It is observed, however, that the order of evaluation has significant effect on the efficiency. Finding an optimal evaluation order is difficult even in the sequential case. In *eager evaluation* all arguments to a function are evaluated before the function is called. Since many imperative languages adopt this strategy for long years, the behavior of functional programs evaluated this way is easily understood in analogy with imperative programs. Another strategy called *lazy evaluation* delays evaluation of every argument of a function until it is required in the body of the function. The *non-strict* functional languages based on lazy evaluation have recently attracted considerable attention in their unique features that are not supported in other languages. Everything has its drawback, however. A weakness of the non-strict languages is the difficulty of reasoning about their space and time behavior [Peyton-Jones87, Chapter 23]. There have many works done for correcting the weakness. For example, *strictness analysis* [Mycroft81] allows the optimization of programs by identifying the parameters that can be evaluated eagerly and avoiding the need to build data structures for lazy evaluation. Program transformation based on lazy evaluation, e.g., [Takeichi87] opens up oppotunities for reducing the resource consumption.

Parallel evaluation of functional programs extends these evaluation strategies. As in sequential evaluation, the evaluation order is irrelevant to the meaning of programs but it has practical importance. One of the most promising works in non-strict parallel functional programming is included in [Hughes84]. Hughes suggests that even on a single processor implementation some form of parallelism is necessary for functional programs to run efficiently. Such quasi-parallelism may improve sequential programs.

In this paper we show how the evaluation partial order of non-strict parallel languages is

This material is based upon work supported by Grant-in-Aid for Scientific Research #01550278.

controlled by synchronization mechanisms. Section 2 introduces primitive functions for parallelism proposed by Hughes [Hughes84]. Experimental results that support his statement are also shown. In Section 3 we give a pragmatic solution for avoiding deadlock that Hughes' synchronization primitive may cause. Implicit synchronization mechanisms inherent to strict functions in lazy evaluation are attractive because they never cause deadlock at all. We show in Section 4 how a higher level notation is translated into parallel code that behaves much like programs using explicit synchronization primitives.

2. Evaluation order and synchronization

Burton proposes a method with which a functional programmer can control the evaluation partial order of parallel programs [Burton87]. The aim of his method is to

- increase parallelism
- reduce storage requirements, or
- reduce the total amount of work performed.

He uses a combination of three parameter passing mechanisms, i.e., *call-by-value*, *call-by-name*, and *call-by-speculation*, which is annotated at the function definition. The call-by-speculation mechanism is considered as an eager form of *call-by-need* that is the default mechanism adopted by non-strict languages. Controlling the evaluation order this way relies on how a function deals with its parameters before the function is called. Omission of call-by-need may cause difficulties in applying this method to non-strict languages where expressions are not evaluated until their values are actually required and expressions are replaced with their values after they are evaluated.

Hughes proposes an alternative method which is simpler and is applicable to non-strict languages [Hughes84]. The expression

```
par e
```

is equivalent to e but evaluated in parallel with the expression containing this construct. Thus

```
f (par e)
```

is semantically equivalent to

```
f e
```

but evaluates e in parallel with applying f to e. The argument passed to the function f might be wholly evaluated or more possibly in a transient state. Hughes call this parameter mechanism *call-by-parallel-evaluation*. As an implementation technique, the *future* structure of Multilisp [Halstead86] might be used. It would be a better idea, however, to take such an argument as a *thunk* with a transient state, because these structures are already present in lazy evaluators.

The primitive par introduces speculative parallelism without any fixed principle. If not used appropriately too many tasks might be spawn to evaluate unnecessary expressions. Strictness analysis [Mycroft81] works in this situation. It derives the information from the program text and decides whether arguments will certainly be required.

Hughes proposes a synchronization primitive synch as a function. The value of

```
synch e
```

is a pair

```
(e,e)
```

except that e will not be evaluated until both the *fst* and the *snd* of (synch e) are required. For example, if we write

```
let (x,y)=synch(1+2) in
          par (fac x) + par (fib y)
```

then we are sure that 1+2 will not be evaluated until both fac and fib are ready to use the result. In fact, if the functions are defined as

```
fac x = if x=0 then 1 else ...
fib x = if x<=1 then 1 else ...
```

then demands are transmitted to 1+2 after two processes reach x=0 and x<=1, respectively. The process that arrives earlier at that point will be suspended until the other arrives. The two processes are thus synchronized according to the demands.

An important application of the function synch is to produce a synchronized list synchlist:

```
synchlist [] = ([],[])
synchlist (x:xs) =
   let (xs1,xs2)=synchlist xs
          and (f1,f2)=synch x in
            ((f1;(x:xs1)),(f2;(x:xs2)))
```

where (f;e) means that evaluate f and wait for evaluation to finish, then return e. The synchronized list is useful for evaluating two functions in parallel and consuming the list elements at the same rate. For example,

```
average xs = (sum xs)/(length xs)
```

may be converted into a parallel program which uses bounded space if we use synchlist:

```
average xs =
   let (xs1,xs2)=synchlist xs in
     par (sum xs1) / par (length xs2)
```

A parallel program obtained by simply annotating par at the two operands of / cannot be evaluated in bounded space; the elements having been consumed by the faster process must remain in the heap until they are used by the slower process.

It should be noted that this method does not alter the program structure, and as far as the value of the program concerned we may ignore par and treat (synch e) as (e,e). It is, however, difficult to decide whether a given parallel program using synch is deadlock-free. We show a practical solution to this problem in Section 3.

To illustrate the effect of the use of synchlist in practice, let us consider the definition of a functional quicksort program. Hughes analyzes this program in detail from a theoretical view point.

```
sort [] = []
sort (x:xs) =
   sort [y|y<-xs;y<x] ++
          (x : sort [y|y<-xs;y>=x])
```

where [e|x<-xs;...] is the list comprehension notation by Turner [Turner85] and ++ is an operator that appends a list to another list. We assume here that list comprehensions are translated into sequential code. Another translation rule will be discussed in Section 4.

A parallel version of the program without synchlist is

```
sort [] = []
sort (x:xs) =
   par sort [y|y<-xs;y<x] ++
     par (x : par sort [y|y<-xs;y>=x])
```

which sparks $3n$ processes to sort n elements. Figure 1 shows a parallelism profile obtained by an execution of the above program written in *PFL* (Parallel Functional Lisp), which is implemented on an ELIS workstation using TAO [Takeuchi86]. Created processes are mapped onto tasks of TAO which are executed by a single processor in turn. Task generation timings are illustrated as if events had been taken place at an equal interval of time and do not represent actual period of time. Instead the ratio of the CPU time is shown at the right. It is observed that most of the time is spent by a few tasks and improvement of runtime would not be expected if many processors become available.

A synchronized version of the quicksort program looks like

```
sort [] = []
sort (x:xs) =
  let (xs1,xs2)=synchlist xs in
    par sort [y|y<-xs1;y<x] ++
      par (x : par sort [y|y<-xs2;y>=x])
```

Hughes analyzes this program and concludes that it will sort n elements in time proportional $O(n)$ using $O(log\ n)$ processors in the best case, or $O(n)$ processors in the worst case. The parallelism profile is shown in Figure 2. It should be noted that the CPU time is shared by many tasks which perform significant amount of work. The total runtime will become much shorter if these tasks are executed on different processors.

Although synch has been devised for making programs use less space, synchronization leads to an effective way of sparking tasks with appropriate granularity. Experiments show that synch may control evaluation partial order of lazy evaluation to some extent in spite of its simplicity.

3. Deadlock-free synchronization

As described in [Hughes84], par introduced in the last section does not affect semantics, but synch may cause deadlock, although it cannot otherwise affect the values computed. Hughes suggests that temporal logic might be appropriate to prove programs free of deadlock. To do so, for a given program using synch, it is necessary to attach logical formula to the program components and then prove the formula for the whole program. An alternative approach to writing safe programs would be to combine par and synch in a way that the program thus obtained are always deadlock-free. We show here how simple application of strictness analysis helps us to find the rules for constructing safe programs.

When we want to write parallel programs, we usually find out several tasks to be performed independently but communicating each other. For simplicity assume that our parallel program is described by two functions f and g both of which take a common argument to communicate. Consider a sequential program

```
H x = F (f x) (g x)
  where f x = ... and g x = ...
```

If we want to use a parallel version, a program like

```
H x = F (par (f x)) (par (g x))
```

would be obtained from the sequential one. In order to evaluate (f x) and (g x) synchronously, we have

```
H x =
  let (x1,x2)=synch x in
          F (par (f x1)) (par (g x2))
```

Is there any possibility of deadlock in executing this program? Yes. If either of the function bodies f and g happens to fail to access the argument, the other function cannot proceed any more.

Strictness analysis [Mycroft81] may be used to determine whether arguments of a function will be eventually evaluated or not. Consider first functions on flat domains such as the integers. We use an abstract domain

$$D^\# = \{\uparrow,\downarrow\}$$

containing two elements. Between the elements $\uparrow$ and $\downarrow$ the order is defined as

$$\downarrow \subseteq \uparrow$$

Moreover, there should be an abstraction function which maps $x \in D$ into $x^\# \in D^\#$ satisfying the property

$$x \subseteq y \text{ implies } x^\# \subseteq y^\#$$

Every non-bottom element in D maps onto $\uparrow$, and the bottom element of D maps onto the bottom element $\downarrow$ of $D^\#$. A function f is strict with respect to its argument x if

$$f^{\#} \downarrow = \downarrow$$

where $f^{\#}$ is an abstract function derived from f. Intuitively this means that if f takes an argument x of which computation will never terminate, then (f x) will never terminate. It is equivalent to say that the value of the argument is certainly required in the body of the function.

In the case of our parallel program, if either of f or g does not satisfy

$$f^{\#} \downarrow = \downarrow$$
$$g^{\#} \downarrow = \downarrow$$

then deadlock cannot be avoided because either of them will not access the argument forever. In order to ensure that the program becomes deadlock-free, we have to do more. Since the equation like

$$f^{\#} \downarrow = \downarrow$$

defines the least fixed point of the relation derived from f, a function definition

```
f x = f x
```

also satisfies the equation of the abstract function $f^{\#}$. In this case, we come across deadlock even if $f^{\#} \downarrow = \downarrow$. What we have to do is to make the strict argument of the function be called by value. We write

```
f (val e)
```

to specify that the argument e is evaluated before f is called. Consequently, we have a deadlock-free parallel program

```
H x =
  let (x1,x2)=synch x in
    F (par (f (val x1))) (par (g (val x2)))
```

provided that f and g are strict with respect to their arguments.

If functions which deal with data structures are considered, strictness analysis on non-flat domains [Wadler87] may be used. Let us take a parallel program using synchlist

```
H xs =
  let (xs1,xs2)=synchlist xs in
    F (par (f (val xs1))) (par (g (val xs2)))
```

As in the case of flat domains, it will become apparent that we have to evaluate strict arguments in order to avoid deadlock. We have already annotated as such in the above program.

For strictness analysis of list processing functions, needed is an abstract domain $D^{*\#}$ for the concrete domain D^{*} of lists of which elements in D. Although Wadler explains how to construct $D^{*\#}$, what we need here is that $D^{*\#}$ should have the bottom $\downarrow$ and an element ∞ corresponding to non-termination and any infinite list, respectively, and that the elements of that domain form a chain

$$\downarrow \sqsubseteq \infty \sqsubseteq \cdots \sqsubseteq \uparrow$$

Strictness analysis described in [Wadler87] tells us to what extent the argument can be safely evaluated. If $f^{\#} \downarrow = \downarrow$, the argument of f may be evaluated before the call; i.e., the function is strict with respect to its argument. Similarly, if $f^{\#} \infty = \downarrow$, it is safe to evaluate the argument and to evaluate the tail of this argument. Wadler says that such a function is *strict in the tail.*

Simple calculation determines whether a function f is strict in the tail. By continuity of the list constructor function cons or the operator : which are assumed non-strict by default,

$$f^{\#} \infty \sqsubseteq f^{\#} (\uparrow{:}\infty)$$

holds. See [Wadler87] for details. Hence we can say that f is strict in the tail if $f^{\#} (\uparrow{:}\infty) = \downarrow$. It should be noted that strictness in the tail implies strictness with respect to the argument.

Now we return to the discussion of our parallel program. As we have done, arguments must be annotated with val for the strict functions f and g. Moreover we have to ensure that the tail of the argument will always be evaluated provided that the functions are strict in the tail. To do this we simply insert val to the tail of the data constructor operator if any as

```
a:(val b)
```

As an example of analysis, consider a higher order function

```
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a xs)
```

This function can be used to define many functions that traverse lists. If we write

$$h^{\#} = (foldr\ f\ a)^{\#}$$

and set $x^{\#} = \uparrow$ and $xs^{\#} = \infty$, we have

$$h^{\#}\ \infty \subseteq f^{\#} \uparrow (h^{\#}\ \infty)$$

To find the least fixed point, we start with

$$h^{\#}_0\, xs^{\#} \ = \ \downarrow \text{ for any } xs^{\#} \in D^{*\#}$$

and iterate substitutions to have

$$\begin{aligned} h^{\#}_{i+1}\infty &= f^{\#} \uparrow (h^{\#}_i\ \infty) \\ &= f^{\#} \uparrow (f^{\#} \uparrow (\cdots\ (f^{\#} \uparrow (h^{\#}_0 \infty)))) \\ &= f^{\#} \uparrow (f^{\#} \uparrow (\cdots\ (f^{\#} \uparrow \downarrow))) \end{aligned}$$

This tells that if

$$f^{\#} \uparrow \downarrow = \downarrow$$

then

```
foldr f a
```

is strict in the tail. That is, if f is strict with respect to its second argument, (foldr f a) is strict in the tail. In that case, a new definition

```
foldr f a [] = a
foldr f a (x:xs) = f x (foldr f a (val xs))
```

may be used to ensure that parallel programs with synchlist are deadlock-free. The functions sum and length used in the average shown in the previous section are defined using foldr as

```
sum = foldr (+) 0
length = foldr inc 0
            where  inc x y = y+1
```

where both (+) and inc are strict with respect to the second argument. Hence the program rewritten as

```
average xs =
  let (xs1,xs2)=synchlist xs in
    par (foldr (+) 0 xs1) /
            par (foldr inc 0 xs2)
```

is also deadlock-free.

It is worth noting that the function map defined as

```
map f [] = []
map f (x:xs) = f x : map f xs
```

is not strict in the tail. Consequently map alone should not be used in synchronized parallel programs. Combinations of map and other functions may become strict in the tail and may appear safely in synchronized parallel programs with annotations inserted as necessary.

4. Implicit synchronization

As a simple example of implicit synchronization of lazy functional programs, consider

```
par (fac 5) + par (fib 10)
```

Two processes are created to evaluate (fac 5) and (fib 10). Although the current process which has created these two processes becomes ready to add two values, it will be suspended until both operands become available. In this way par and strict functions like + achieve synchronized parallel evaluation. Such a synchronization mechanism is different from that of par and synch, however. If we understand that par behaves as the *fork* operation of imperative parallel programming, synchronization by strict functions is considered as the *join* operation.

In this section we show how implicit synchronization works as the synch function. One of the advantages of implicit synchronization is that it does not require the programmer to do a subtle business of inserting synch in the right place. Moreover, the program does never cause deadlock which is the most annoying problem of the synch primitive.

It is not true that an arbitrary functional program would run efficiently in parallel. The program to be executed in parallel must contain *algorithmic parallelism.* In modern functional programming such inherent parallelism should be specified in higher level notations. One of such notations is a list comprehension, or a set notation by Turner [Turner85]. The expression

```
[f x| x<-xs]
```

specifies the list consisting of

$$f\ x_1, f\ x_2, \cdots, f\ x_n$$

if xs stands for

$$[x_1, x_2, \cdots, x_n].$$

It would be easy to understand that such an expression contains inherent parallelism; every $f\ x_i$ may be evaluated in parallel with each other. Of course, we have to assume that xs should represent a finite list.

Translating comprehensions into combinations of map, filter, and concat is carried out according to the rules [Bird88, p.63]:

```
(1) [x|x<-xs] = xs
(2) [f x|x<-xs] = map f xs
(3) [e|x<-xs;p x; ...] =
        [e|x<-filter p xs; ...]
(4) [e|x<-xs;y<-ys; ...] =
        concat [[e|y<-ys; ...]|x<-xs]
```

The functions used in the translation are assumed to be evaluated sequentially.

```
map f [] = []
map f (x:xs) = f x : map f xs

filter p [] = []
filter p (x:xs) =
        x : filter p xs, p x
        filter p xs, otherwise

concat [] = []
concat (xs:xss) = xs ++ concat xss

[]++ys = ys
(x:xs)++ys = x:(xs++ys)
```

Observing that the rule (2) translates inherent parallelism into sequential list manipulation, we replace it with a parallel version:

```
(2') [f x|x<-xs] = parmap f xs

parmap f [] = []
parmap f (x:xs) =
        par (f x) : val (parmap f xs)
```

This introduces parallelism in a straight way. Then, from where does implicit synchronization come out? In the above transformation, we do not use any strict functions at all except parmap that spawns processes in turn. The reason why strictness arises in the program is that a strict function like show to see the value on the terminal is usually applied at the top level of the program. Strictness of the top level function is propagated to the function concat where actual synchronization takes place.

As an example of the use of a parallel comprehension, we show execution profiles of two programs for the 5-queens problem. Both programs are contained in [Bird89]. The first program looks like

```
queens 0 = [[]]
queens (m+1) =
 [p++[n]|
   p<-queens m;n<-[1..5];safe p n]
```

where safe is implemented directly by a function in our program. Bird uses a list comprehension to define safe. Another version of this problem is

```
sneeuq 0 = [[]]
sneeuq (m+1) =
  [p++[n]|n<-[1..5];p<-ps;safe p n]
                        where ps=sneeuq m
```

Figures 3 and 4 illustrate execution profiles. The root task (the topmost process) consumes large portion of the total CPU time because the top level function show need to transmit demands to the program body. Putting this aside, remaining CPU time is shared by many tasks much the same way as in Figure 2. We may say that implicit synchronization works well in the parallel list comprehension.

5. Remarks

We have examined the property of the synchronization primitive synch in detail. Although Hughes originally introduced synch in order to make parallel programs run in bounded space, it appears that the use of synch may bring out a desirable tasking mechanism for actual parallel functional systems.

In order to write deadlock-free programs which contain synch for synchronization, we have proposed a practical method to avoid deadlock by building up programs from safe functions. It is somewhat informal for brevity, however. More formal treatment would be welcome.

Synchronization by strict functions is inherent in lazy evaluation and is commonly observed in programs that deal with numbers. It may work to some extent for list processing functions as well. We have demonstrated this by a parallel implementation of the list comprehension. There is more work to be done before it can become a higher level notation of parallelism.

References

[Bird88]Bird, R., and Wadler, P.: *Introduction to Functional Programming*, Prentice-Hall, 1988.

[Burton87]Burton, F.W.: Controlling Reduction Partial Order in Functional Parallel Programs. *Graph Reduction*, LNCS **279**, Springer-Verlag, 240-251, 1987.

[Halstead86]Halstead, R.H. Jr.: An Assessment of Multilisp: Lessons from Experience. *International Journal of Parallel Programming* **15**, 459-501, 1986.

[Hughes84]Hughes, R.J.M.: *Parallel Functional Programs Use Less Space*, Programming Research Group Memo, Oxford University Computing Laboratory, 1984.

[Mycroft81]Mycroft, A.: *Abstract Interpretation and Optimising Transformations for Applicative Programs*, PhD thesis, University of Edinburgh, 1981.

[PeytonJones87]Peyton Jones, S.L.: *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

[Takeichi87]Takeichi, M.: Partial Parametrization Eliminates Multiple Traversals of Data Structures. *Acta Informatica* **24**, 57-77, 1987.

[Takeuchi86]Takeuchi, I., Okuno, H., and Ohsato, N.: A List Processing Language TAO with Multiple Programming Paradigms. *New Generation Computing* **4**, 401-444, 1986.

[Turner85]Turner, D.A.: Miranda: A non-strict functional language with polymorphic types. *Functional Programming Languages and Computer Architecture*, LNCS **201**, 1-16, Springer-Verlag, 1985.

[Wadler87]Wadler, P.: Strictness Analysis on Non-flat Domains. *Abstract Interpretation of Declarative Languages*, 266-276, Ellis Horwood, 1987.

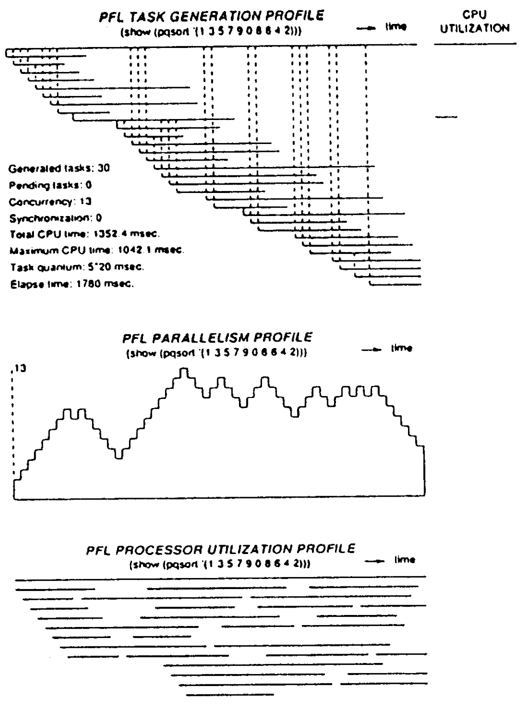

Figure 1. Parallel Quicksort

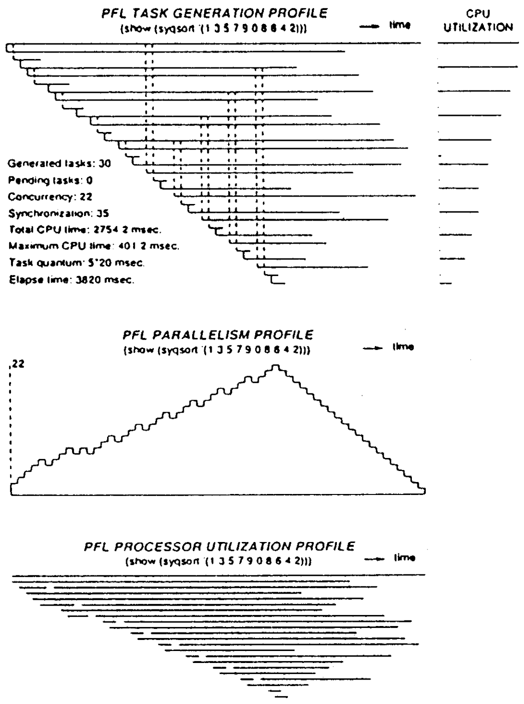

Figure 2. Synchronized Quicksort

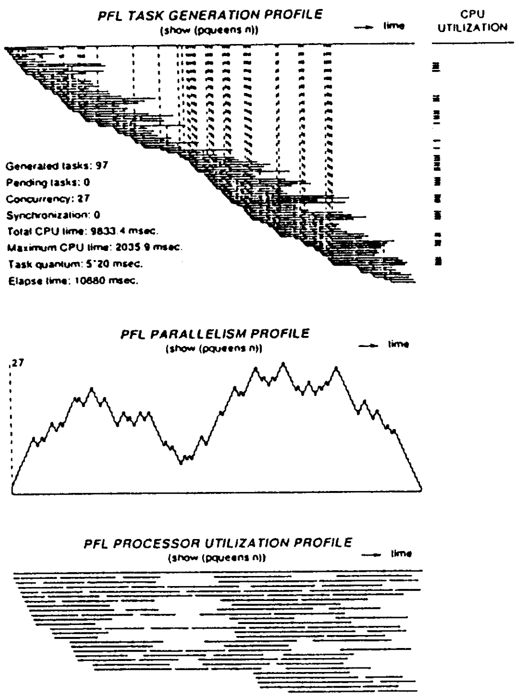

Figure 3. queens 5

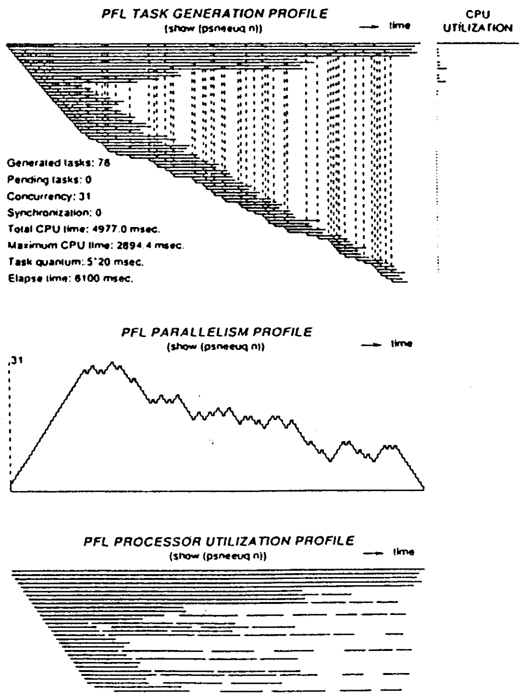

Figure 4. sneeuq 5