Concurrency and Synchronisation in Actel

Zair Abdelouahab
Universidade Federal do Maranhão
Departamento de Engenharia de Eletricidade
Centro Tecnologico
Campus Universitario do Bacanga
São Luis 65080-400
MA- Brasil
E-mail: zair@fapema.br
Tel: +55 98 232 34 66 Ramal 181
Fax: +55 98 232 26 18

Peter M Dew
School of Computer Studies
University of Leeds
Leeds LS2 9JT
E-mail : dew@uk.ac.leeds.scs
Tel : +44 - 532 - 335432
Fax : +44 - 532 - 335468

Abstract

This paper introduces concurrency mechanisms of a new concurrent object based language called Actel. In particular, it focuses on the issues of how to explore parallel computation with object oriented techniques, how to achieve a good run time efficiency, and how to avoid the burden of explicit synchronisation.

Actel offers a variety of inter and intra-object mechanisms to exploit concurrency at several levels of an object. A new mode of message passing called `semi-reference' is devised to achieve an efficient inter-object communication and to efficiently support inter-object parallelism (coarse grain). The semi-reference allows transmission of references within messages to achieve an efficient delegation whilst preventing inconsistencies. Parallel function are allowed to execute inside an object to achieve medium grain of parallelism without recourse to implicit synchronisation. Finer grain of parallelism can be obtained by activating parallel compound statements inside parallel functions. Multiple future variables (simple or overloaded) are provided to remove the burden of explicit synchronisation, to be the place holder of results from the parallel functions and compound statements and to maximise parallelism.

1. Introduction

Over the last decade, the concurrent object oriented programming methodology has become popular. Programs are usually expressed as a set of independent and concurrent modules called objects that interact via message passing. Message transmission may take place concurrently among objects, enabling these to be active in parallel and giving rise to inter-object concurrency. Upon receipt of a message, an object becomes active and executes the appropriate function (or method) as specified by the received message. It may also spawn multiple, internal threads of execution, giving rise to intra-object parallelism.

The motivation of our study has been to generalise the object oriented techniques used in the design of parallel solid modelling systems developed at Leeds [7] (e.g. Mistral-3). Shared Objects (SO) [10] is another system developed at Leeds as a highly parallel programming environment. Our work has been to investigate the usage of SO as a target system.

Our survey of existing concurrent object oriented languages highlightned a number of weaknesses which we have attempted to overcome in a new language called Actel. This has been designed for high performance computers and in particular

- It introduces a new message passing mode (semi-reference) to improve the efficiency of languages that use delegation.
- It explores a number of programming constructs for expressing parallelism at different levels of granularity within an object
- It supports implicit synchronisation to releave the programmer from the burden of explicit synchronisation whenever multiple threads of activities are allowed within an object.

2. Review of Concurrency and Synchronisation

2.1. Overview of Inter-Object Concurrency

Inter-object concurrency, with message passing as the means to effect synchronisation and control flow, leads to fairly simple and elegant parallel programs. However, there is a performance overhead incurred with each transmission of a message. This overhead becomes significant when delegation is used, since the message may be re-transmitted several times before it arrives at the final destination. A possible solution is to employ message passing by reference, in which only the address of the message is initially transmitted---the rest being sent directly to the final destination. This overcomes the inefficiencies associated with delegation, but can introduce data inconsistencies; for example, when the source object updates the message before it is dereferenced at the destination. In this simple form, message passing by reference violates the object oriented rule of encapsulation.

2.2. Overview of Intra-Object Concurrency

Intra-object concurrency refers to having more that one thread of activity within a single object. Synchronisation is required to protect the shared state of the object and ensure that it remains consistent. Control of concurrent access to the state is usually handled by the object itself, based upon a centralised or decentralised algorithm

Centralised Synchronisation : In this model of synchronisation a single central controller or procedure governs which method should execute and therefore gain access to the shared state. Concurrent object languages based on this model are ABCL [13], POOL [2] and the Extended

Eiffel [4]. In POOL, only one thread of execution and one activity may be started at any time; in ABCL, only one thread may be executing, but many activities may have started and suspended---perhaps due to an express request. Languages based on this model of concurrency restrict parallelism.

Decentralised Synchronisation : In the decentralised model of synchronisation, control is distributed amongst the methods rather than being centralised in a single procedure. There are a variety of decentralised synchronisation algorithms, of which some utilise explicit synchronisation techniques and others implicit or automatic techniques.

Synchronisation using critical regions: COOL [5], Concurrent Smalltalk [12], Trellis/Owl [6] are languages that use critical regions. Multiple threads are allowed to execute concurrently, but are required to explicitly lock and unlock shared variables. Whilst this ensures consistency of state, it also complicates the program. COOL addresses the problem with future variables---predefined variables which contain the return values of functions, but block an accessing process if the variable has not yet be assigned. Unfortunately, COOL's simple future mechanism is limited to one future variable per parallel function, any other values must be returned through explicit synchronisation of the objects state.

Synchronisation using special monitors: COOL is a language that uses a special monitor called *mutex*. A method associated with the mutex attribute is allowed to execute exclusively within an object. Other methods are prevented from running until the currently executing method has terminated and released the monitor. This form of synchronisation leads to deadlock when the monitor is not released.

Use of replacement behaviour: Actor languages such as Act2 [11] and Act3 [1] use this type of implicit synchronisation. An actor (or active object) creates a replacement actor which runs concurrently with its creator and performs the rest of the computation. Synchronisation is automatic because the two actors now have different states. Since actors are objects with fine granularity, the efficiency of this model on current machines remains in doubt since the cost of communication remains expensive.

3. Message passing by Semi-Reference

Inter-object concurrency in Actel is achieved using asynchronous message passing to maximise parallelism. Three modes of message passing are provided; the conventional modes of by value and by priority (messages sent with a priority value), and a new mode for delegation, by semi-reference. In this paper, details are given only on the semi-reference mode of message passing.

The semi-reference mode of message passing in Actel is based on sending references of data within the message rather than the data itself. Modes of message passing based on reference transmission usually lead to improved run time efficiency. However, simple reference mechanism is unsuitable since it leads to violation of encapsulation and data inconsistencies. In Actel, a refined reference technique *semi-reference* is employed in which access to the referenced data is strictly controlled.

In the `semi-reference' communication mode, Actel allows the object to acquire a reference or message data using a specific referencing mechanism. Prior to sending the message the

source object has control of the reference; after the message is sent, the source loses this control. The destination object gains control of the reference upon receipt of the message. It may dereference the message and thus obtain the data, or send the reference to another object. The referencing technique is shown in figure 1.

Constructs	Meaning
ref = |> data	acquiring a reference for data
data = <| ref	data acquires the content referenced by ref.

Figure 1: Referencing and Dereferencing Mechanisms in Actel

In the semi-reference model only references are transmitted. Data is actually transferred only when the message is dereferenced at its final destination. This mechanism is particularly efficient when one object delegates work to a second, and this to a third, since only the reference needs to be forwarded by the intermediate objects.

4. Intra-Object Concurrency and Synchronisation in Actel

Actel supports intra-object concurrency in several ways: by activating parallel functions inside objects; by invoking parallel functions inside parallel functions; by allowing parallel compound statements to run inside parallel functions; and by executing several methods of multi re-entrant (stateless) objects simultaneously.

4.1. Parallelism Using Parallel Functions and Multiple Futures

Similar to COOL [5] and Parallel Eiffel [4], Actel parallel functions are spawned by a function call in which future variables are defined to hold the function return values. One future is assigned to the parallel function return value, others hold parameter return values (i.e. futures passed as parameters to the parallel function to hold returning results). With each parallel function it is therefore possible to associate 'multiple futures'. The calling process (or creator) continues in parallel with the spawned parallel function. When the creator requires one or all of the parallel function results, it blocks pending assignment of the future variables. Methods in Actel are able to block on specific future variables, and therefore only need to wait for partial results from the parallel function. This means that a method may access results of a future variable defined as parameter while the function is still running. This mechanism, which is specific to Actel, maximises internal parallelism.

Results of parallel functions are returned only through future variables; this has an advantage over COOL since it relieves the programmer from the burden of explicit synchronisation. In COOL, the only way to access partial results of a parallel function is through their shared state and this is done at a cost of explicit locking and unlocking. The latter mechanism may be viewed as a complication to the program.

Example Using Multiple Futures

To illustrate the use of multiple futures, a piece of Actel code is presented in figure 2.

An object 'A' receives a message to carry a computation using a method 'compute'. When the method is processing, a separate thread of control is created through a parallel function func_a

to perform a part of the computation. The object and the function are therefore executing in parallel. When the function terminates it returns an integer result which is assigned to the future variable `fut_var'. Whenever, the object needs the result, it blocks at the future variable `fut_var' using `receive'. The content of the future variable is then transferred to the integer variable `res'. In order to increase parallelism, a future variable `fut1' is passed as a parameter to the function. The future variable holds the partial result returned by `func_a' through the `reply' construct. When `fut1' is needed by object `A', it is accessed and used while the function is still running. The access is done without recourse to explicit synchronisation.

```
begin object : fred
state (
int a,b,res ;
future fut1, fut_var ;
)
begin methods
    accept compute with a b
    do
        /* call a parallel func\_a */
        fut_var = func_a(a,b,fut1) ;
        /* do some more computation */
        ....
        /* access the future variable fut1 */
        b = receive(fut1);
        ....      /* use b */
        res = receive (fut_var) ;

par_functions
    end
    ....
end methods
```

```
begin par_functions
    int func_a(a,b,fut1)
    int a,b ;
    future fut1 ;
    {
    int x ;
    /* make changes to b */
    ....
    ....
    reply b to fut1;
    ....
    /* compute x using a and b */
    ....
    /* return x as the function value */
    return (x);
}
..... /* use res subsequently */          end

end object fred
```

Figure 2: Parallelism Using Parallel Functions and Multiple Futures

Programming this example with COOL would be more complicated (since explicit synchroni-

```
int func_fred(...)
...
{
    int $fut_var ;
    /* call a parallel func_a */
    fut_var = func_a (a, &b);
    /* do some more computation */
    ....
    /* Access the variable b */
    /* Use b subsequently */
    ....
    ....
    return (b);
}
```

```
int func_a(a,b)
int a,*b ;
{
    int x ;
    ...
    /* make changes to b */
    lock (b) ;
    /* set the value of b */
    unlock (b) ;
    /* Compute x using a and b */
    ....
    /* return x as the function value */
    return (x) ;
}
```

Figure 3: Parallelism With Parallel Functions in COOL

sation through mutex or lock variables is required) or less efficient since some of the parallelism is lost by not obtaining partial results. This is the case of the program presented in figure 3. The parameter *b* is passed by reference using the C reference model from *func_fred* to *func_a*. The function *func_fred* cannot access *b* before the function *func_a* returns since it may create an inconsistency. The locking mechanism used in this example is not even useful in this case because it is not known which process first accesses *b* (either *func_a* or *func_fred*). The safest way to use *b* is after the function completes (i.e. by first blocking on the future variable *fut_var* and then access b)---this results in a loss of potential parallelism.

4.1.1. Future Variables

In Actel, future variables are defined using the type constructor *future*. They are specialised variables which may store data of different types; for example integers, floats, chars, arrays, and complex structures. The type of a future variable is bound to the type of its value (that is, at run time). Future variables provide a global abstraction since they refer to locations where the data is stored. They may be passed to other objects or to other functions, causing a synchronisation between each process that blocks when accessing them. If it contains a value, the future variable is said to be *set* (or *resolved*); otherwise it is said to be *unset* (or *unresolved*). A future variable is set using the *return* or the *reply* statements of the parallel thread (e.g. parallel function). When the future becomes resolved, all processes which are blocked waiting with a *receive* operation for the value are resumed.

4.2. Parallelism Using Asynchronous Compound Statements

Actel supports another level of parallelism based on creating parallel compound statements inside functions to exploit a finer grain of parallel computation. As with parallel functions, the invocation of one or more compound statements lead to creation of asynchronous and parallel threads. The calling thread associates one or more futures to hold return results of the invokee. Results of the parallel threads (or parallel compound statements) are accessed in the same way as with the parallel functions; that is, with blocking pending assignment on the future variables. A parallel compound statement is defined using one of the following forms inside parallel functions.

Form-a

```
withfutures (future_list) {
        block statement
}
```

Form-b

```
withfutures (future_list) withparameters} (parameter_list) {
        block statement
}
```

where the *block statement* describes the body of the parallel compound statement (composed of declarations and constructs). *withfutures* is an Actel keyword used to define a list of future variables passed as parameters to the parallel compound statement to hold returning results. Other parameter types (e.g. C parameters) are either passed implicitly (form-a) or explicitly using the expression *withparameters* (form-b). In the former, any variable used and not declared within the body of the parallel compound statement is automatically defined by the compiler inside the parallel thread and its value is passed from the parallel function. In (form-

b), the list of parameters which are passed to the thread are given by the expression *parameter_list*. This mechanism is more efficient since it does not require the compiler to search for variables that are not declared in the body of a compound statement. The component *future_list* defines a list of future variables. The first future variable of the list is termed the *master future* because its value is returned from the parallel statement using a C *return* statement. Other future variables are set by the parallel compound statement using the *reply* statement shown in section 4.1.

4.2.1 Simultaneous Parallel Compound Statements

In Actel several threads may be created simultaneously using a C iterative statement. Its general form is:

```
for (i=0; i<n; i++)
        withfutures(mfut[i]::future_list) {
                block statement
        }
```

Its effect is to create *n* parallel threads executing simultaneously and in parallel rather than one by one. Each thread is provided with future parameters to maintain results; for example, *mfut[i]* is the master future used to store the value of a thread *i* whereas the variables in *future_list* hold intermediate results of each thread. Futures variables of *future_list* are overloaded since several threads may return several results to the same location. The extra space used by these futures can be allocated dynamically and removed whenever they are not required. This leads to improved utilisation of the memory space.

To access an overloaded future variable, Actel provides a mechanism which distinguishes between different values of an overloaded future. It has a general form:

```
var = receive(mfut[i] ::fut);
```

Each value of a future variable can be identified by the thread with which it is associated at the time of the call. For instance, the value of the overloaded future *fut* returned by thread *i* is given by the expression *mfut[i]::fut*. The expression *var* indicates the local variable to which the value of the future is to be copied. The expression *mfut[i]* is the master future corresponding to the thread *i* and :: is the overloading operator.

4.3. Parallelism Using Parallel Functions Inside Parallel Functions

Actel may express another form of parallelism inside a function; that is a function executing on an object may invoke other parallel functions or itself. The technique has been employed by COOL to express its finer granularity of parallelism; but in Actel, it is still possible to express more fine grain computation with asynchronous compound statements which are shown in section 4.2. The example presented in figure 3 shows a parallel function *func_fred* of COOL making a call to *func_a*. In Actel, such call can be made in a similar fashion.

4.4. Parallelism With Multi Method Execution

In Actel several methods execute simulataneously and in parallel inside multi re-entrant objects without a need to explicit synchronisation. The technique has been employed by some languages such as Emerald [3], but explicit synchronisation is usually required.

4.5. Manipulation Of Future Variables

Future Variables Usage:

A future variable may be used by any object, parallel function or parallel compound statement which refers it. It may be set by a *return* or *reply* construct of a parallel function or a compound statement. It may also be re-set by any one of the above constructs if necessary. However, re-setting a future variable will affect every thread which refers it. To avoid ambiguities, Actel provides mechanisms which allow reuse of future variables in a more consistent fashion. This is done in two ways. In the first, the future can be reused without affecting other processes which refer it. This can be achieved by clearing the location locally from an object or a parallel function using:

clearloc(fut);

As a result of this operation, the location found inside the future variable *fut* will be cleared and a new location is assigned. The future variable *fut* can therefore be passed to other objects or function, whereas the old location can still be used by other threads.

In the second, the value of a future variable is cleared. That is, the content of the location is cleared. The future then becomes *unresolved* and all the processes block until it is set by a new value by the function. The operation for clearing a future variable has the form:

clearfut(fut);

Parallel Access to Future Variables:

When parallel functions and parallel compound statements are activated, they run asynchronously. Future variables associated at the time of the call with the parallel threads become resolved when these terminate. However, it is not known which future is set first since the execution of the threads is undeterministic. An access to a future variable is performed with a blocking *receive* operation. Therefore, accesses to several future variables by a single threads can be done only sequentially and may generate delay. To avoid delay, Actel provides a *parloop* mechanism which permits to block on several futures in parallel. It has the form:

```
begin parloop
        var1 = receive(fut1) {
                block_statements_1
        }
        var2 = receive(fut2) {
                block_statements_2
        }
end parloop
```

For any future *fut1* or *fut2* which become set, its value is accessed and its corresponding *block_statements_1* or *block_statements_2* is executed. The parloop statement terminate when all the future variables have been set.

4.6. Compared to Other languages

Languages can be characterised by the concurrency model upon which they are founded (e.g. inter and intra-object). For example, ABCL and POOL are languages that supports only inter object concurrency. While these languages follow a simple model, they are less expressive since they require the programmer to structure an application as a set of objects that work

cooperatively. Actel, on the other hand, supports both inter and intra-object concurrency and has the potential of expressing parallelism more naturally. COOL and Emerald provide support for intra object concurrency, but require an explicit synchronisation which may constrain the amount of parallelism. In Actel, synchronisation is implicit and is obtained with message passing or future variables.

Parallel functions are the sole mechanism in COOL for achieving concurrency (coarse to fine grain) whereas Actel provides several mechanisms including parallel compound statements for an efficient fine grain parallelism.

Future variables in Actel are different from those provided by COOL. In particular, they refer to locations where the data is stored. They provide support for polymorphic type to store data of any size and any type. In addition, they may be overloaded in which case they can store one or several results. Future variables can be passed from objects to parallel functions, from parallel functions to parallel functions and from parallel functions to parallel compound statements to achieve synchronisation and maximise parallelism. In Actel, accesses to future variables can be performed in parallel whereas in COOL it can be achieved only sequentially. In COOL, a future variable associated at the time of the call of a parallel function may become garbage if the caller exits without waiting (i.e. blocking) on the future variable. In Actel, this problem does not occur because future variables provide a global abstraction.

5. Implementation

There are fundamentally two types of multiprocessor systems: tightly coupled shared memory systems and loosely coupled distributed memory systems. Shared memory systems provide a single address space accessible by all processors, while distributed memory systems provide each processor with its own private address space and sharing of data takes place with explicit message passing [10]. An alternative approach is to use a higher level of abstraction that can be mapped efficiently on to either of the above machines. This is provided by the Shared Objects (SO) environment [10]. SO supports a model for inter-process communication through shared abstract data types (e.g. queue, priority queue, bag, and stack) where user processes are allowed to read and write these concurrently. An experimental system is available at Leeds for a transputer network.

Actel has been implemented using SO and is currently running on a Meiko computing surface. The implementation is based on the development of a translator that takes Actel code as input and produces a C program exploiting SO functions. The target code is directed at a multiprocessor machine(e.g. the transputer network). This output consists of a set of parallel processes or parallel servers which are mapped on to different processors at load time. They exploit the user code and provide the functionality of the Run Time System (RTS). The major issues of Actel implementation are: the RTS, the semi-reference mode of message passing, the parallel threads and the future variables.

5.1. The Run Time System

The major responsibilities of the Actel Run Time System are: creation and management of the dynamic objects, handling communication and buffering, process task creation and process task distribution. Tasks are created as a result of: creation of objects, invocation of parallel functions, activation of parallel compound statements and communication. Each of these

created tasks is allocated sufficient resources and queued for processing at appropriate server. The tasks in Actel are implemented as messages to request from servers the appropriate operation

Functions of the Actel Run Time System are implemented within the server processes. There are three types of server processes. The first process is called the *process driver* and contains method code. The second process is known as the *function server* and holds parallel functions code of an object type (i.e. class). The third process type termed *compound statement server* has the code of a single parallel compound statement of an object type. All of these processes are replicated on every node of the parallel machine.

Each process server has a well defined role. For instance, each process driver manages the objects created or removed in its workspace, objects scheduling and descheduling for processing and task management. Function server processes manage and schedule tasks related to parallel functions for execution. Process compound servers are responsible for executing tasks of appropriate compound statements. These processes also make use of the target code of an application program generated by the translator.

5.1.1 Task Management and Distribution

Tasks in Actel are classified into two categories: specific and generic. Specific tasks refer to tasks generated as a result of communication to specific objects. Generic tasks refer to those which can be executed on any replicated process server (e.g. parallel function tasks, compound statement tasks and multi re-entrant object tasks). Generic tasks are stored in shared abstract data types and are accessible by every process server. In this way, load balancing and distribution can be achieved more efficiently. Specific tasks, on the other hand, are stored in queues that are accessible only by specific servers on that processor.

5.2. Implementation of Future variables

Since future variables are locations for holding returning results which might be accessed by several processes, it is therefore more suitable to implement them as shared structures. For this purpose, these are implemented as shared abstract data types provided by SO. Accessing the future variables corresponds to the process of reading from the shared abstract data types. This may happen only if the future variables have returned (i.e. results are written to the shared abstract data types). Attempts to read a variable before it is set are blocked by SO. SO provides data synchronisation by suspending processes if necessary.

5.3. Implementation of Message Passing

Since SO provides an inter-process communication through shared abstract data types, the implementation of the semi-reference message passing is straightforward on a simple message passing system that provides a write to destination. The implementation requires message references rather than content to be transmitted, and supervisory code to ensure that only one active object owns the reference at any one time. This is achieved by making a copy of the data and storing it in a shared abstract data type. The address of the shared data is then sent to the destination. At the destination, the data is read from the shared abstract data type.

5.4. Performance Results

Experiments have been compiled to measure the performance of the semi-reference mode of message passing and the mechanisms of activating parallel functions and parallel compound statements.

In the first set of experiments, a comparative study between the semi-reference and the value mode of message passing is undertaken. First, a message is sent from node zero to another node whose identifier is one using the value and the semi-reference modes of message passing. The measurements are taken in microseconds and shown in figure 4. The results show that the value mode performs better than the semi-reference in case of transmission directly from source to destination. This is obvious since the cost for sending a message by semi-reference includes the cost of referencing and dereferencing a message.

Value	912
Semi-Reference	1508

Figure 4: Cost of a Single Integer Transmission

In another experiment, the same message is passed between an object located on a node identifier 0 to another on node 1. The latter delegates the message to a third object on node 2, then to a fourth on node 3, and so on. The same experiment is applied to networks of 4, 8, 12, 16, 20, 24, 28 and 32 processors T800.

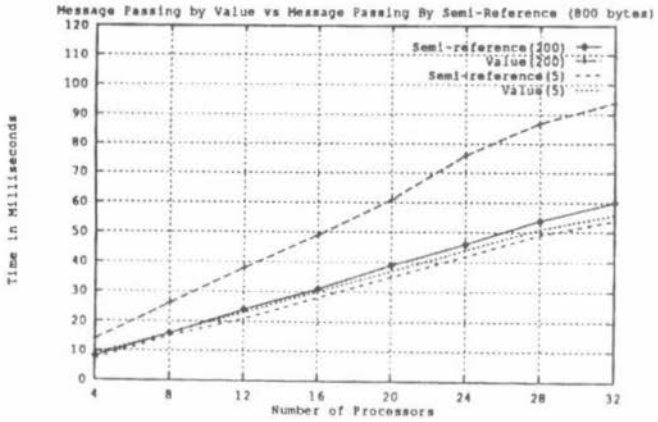

Figure 5: Hopping 5 and 200 Integers

The initial object records the average time to send a message from its source to its final destination. The timings are collected in microseconds for messages of 5 and 200 integers and are shown in the graph of figure 5. These results do not include the overhead of the underlying system (SO). The cost of hopping by value 5 (200) integers is between 1 to 1.2. (1 to 2) higher than the cost of hopping them by semi-reference. In general the results show that the semi-reference is more efficient than the value and that the efficiency increases with respect of the message size.

In the second set of experiment, a comparison between different parallel threads is also undertaken. This experiment is conducted in three steps. First, 300 multi re-entrant objects are created and sent communications to compute tasks of 100 ms each; their results are returned also through messages. Second, the same number of parallel functions are created to compute similar tasks; their results are accessed pending assignment on the future variables. Finally, the same experiment described in step 2 is applied for the parallel compound statements. In each step, the initiator records the timings for the whole process to complete for different sizes of the network of processors. Results are shown the graph of figure 6.

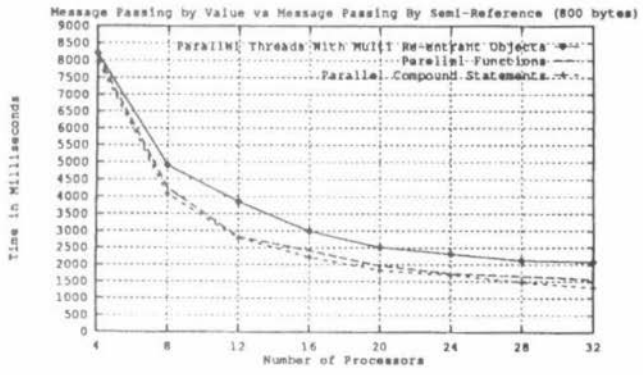

Figure 6: Parallel Threads Activation

The graph shows that the performance in each component increases in proportion with the size of the network. They also show that the cost of using multi re-entrant object is the highest of them. This is expected since it involves creating objects, sending communications to compute the tasks, and waiting for the results to return through messages. The cost given by the parallel compound statements is slightly better than the cost of parallel functions because they compute similar tasks. However, it should be more advantageous to compute smaller tasks with the parallel compound statements since these tasks are executed before those of parallel functions. These results are good for Actel since it shows that intra-object concurrency with parallel functions and parallel compound statements is cheaper than inter-object concurrency using message passing.

In the same set of experiments, accesses to the future variables associated with parallel functions are assessed. The future variables have an index ranging from 1 to 300. Accesses are made with the parloop and without the parloop statement using the order from 300 to 1. Results are shown in figure 7.

Clearly, the results show that the usage of parloop construct reduces the delay incurred when accessing future variables since the blocking is performed in parallel. Specifically, the reduction of the delay is apparent on a small network of processors (i.e. where contention exists).

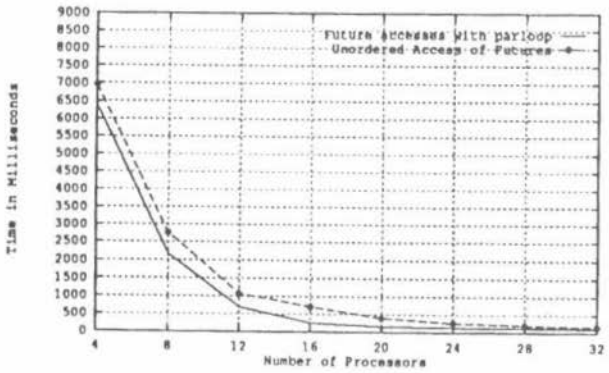

Figure 7: Future Variables Accesses

6. Summary and Further Development

This paper has presented the mechanisms of inter and intra-object concurrency in Actel. In particular, it introduces a new mode of message passing `semi-reference' to achieve an efficient delegation (inter-object concurrency). It uses parallel functions as a means to obtain concurrency inside objects (medium grain). It supports the concept of `parallel compound' statement to achieve a more efficient finer grain of concurrency inside parallel functions. It introduces the concept of `multiple future variables' (simple and overloaded) combined with parallel functions and parallel compound statements to maximise parallelism and achieve an implicit synchronisation; thus removing the burden of an explicit synchronisation.

In the design of the language, a considerable attention has been paid to include features which will generate a good run time efficiency. In particular, experiments have shown that message passing by semi-reference is useful for hopping messages between several objects. Results have also shown that small granularities of parallelism can be achieved more efficiently with parallel compound statement than with parallel functions or object activation.

Further development will concentrate, in particular, to an evaluation of the concurrency mechanisms using a variety of applications.

7. Acknowledgements

We should like to thank members of the Parallel Processing Group at Leeds University for their useful comments and discussions. Financial support of Fapema is gratefully acknowledged.

8. References

1. Agha, Actors: *A model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge Massachusetts USA 1987.

2. America, *POOL-T: A Parallel Object Oriented Language*, In A. Yonezawa and M. Tokoro, editors, Object Oriented Concurrent Programming, pages 199-220, MIT Press 1987.

3. Black et al, *Object Structure in the Emerald System*, In OOPSLA'86, ACM Sigplan Notices, vol 21 (11), pages 60-65.

4. Caromel, *Concurrency: An Object Oriented Approach*, In TOOLS PACIFIC'90, Sydney, November 1990.

5. Chandra et al: *COOL: A Language for Parallel Programming*, Chapter 8, In Gerlenter et al, 1990.

6. Elliot and J. Moss, *Concurrency Features for the treuillis/Owl Language*, In J. bezivin et al, editors, ECOOP'87, Lecture Notes in Computer Science, Springer-Verlag, 1987.

7. Holliman, C. M. Wang and P. M. Dew, *Mistral-3: Parallel Solid Modeling*, The Visual Computer for a Special Issue on SuperComputing for Visualisation, 1992.

8. kafura and K. Lee: *Inheritance in an Actor Based Concurrent Object Oriented Programming Languages*, In Stephen Cook, editors, ECCOP'89, pages 131-146, Cambridge University Press.

9. Lieberman, *Concurrent Object Oriented Programming in Act1*, In A. Yonezawa and M. Tokoro, editors, Object Oriented Concurrent Programming, pages 9-36, MIT Press 1987.

10. Mallon and P. M. Dew, *Communicating through Shared Objects*, In Proc of IFIP Conference on Programming Environments for Parallel Computing, Edinburgh 1992.

11. Theriault, *Issues in the Design and Implementation of Act2*, Technical Report TR-728, MIT AI Laboratory, 1983.

12. Yokote and M. Tokoro, *Concurrent Programming in Concurrent Smalltalk*, In A. Yonezawa and M. Tokoro, editors, Object Oriented Concurrent Programming, pages 129-158, MIT Press 1987.

13. Yonezawa et al, *Object Oriented Concurrent Programming in ABCL/1*, In OOPSLA'86, ACM Sigplan Notices, vol 21 (11), pages 258-268.