Using Optimistic Execution Techniques as a Parallelisation Tool for General Purpose Computing

Adam Back and Stephen Turner

Department of Computer Science, University of Exeter,
Prince of Wales Road, Exeter, EX4 4PT, England
Email: {aba,steve}@dcs.exeter.ac.uk
Tel: +44 1392 264048 Fax: +44 1392 264067

Abstract. Optimistic execution techniques are widely used in the field of parallel discrete event simulation. In this paper we discuss the use of optimism as a technique for parallelising programs written in a general purpose programming language. We present the design and implementation of a compiler system which uses optimistic simulation techniques to execute sequential C ++ programs. The use of optimistic techniques is seen as a new direction in parallelisation technology: conventional parallelising compilers are based on static (compile–time) data dependency analysis. The reliance on static information imposes an overly restrictive view of the parallelism available in a program. The static view must always be the worst case view: if it is possible for a data dependency to occur, then it must be assumed always to occur.

1 Introduction

Optimistic execution schemes such as *Time Warp* [5] have been successfully used for parallel discrete event simulation (PDES). The aim of our research is to investigate the use of optimism as a technique for parallelising programs written in a general purpose programming language. The motivation for using optimistic techniques in parallelising code stems from some of the restrictions associated with compilers based on static data dependency analysis. In compilers using static data dependency analysis only, the program is analysed to determine independent sections of code. If two sections of code have no data dependencies between them they may be executed in parallel. This observation forms the basis of static analysis based parallelisation. When applied to looping constructs static analysis is used to determine if different iterations of the loop are independent. If they are independent then the iterations may be executed in parallel.

This technique works where the analysis can determine that a reasonable number of sections of the program are independent; however, there will be applications where the static analyser will fail to find sufficient numbers of independent sections. At this point there are two possibilities: either the program is not parallelisable, or static analysis is not sufficient to discover the parallelism.

There is nothing which we can do about pathologically unparallelisable code, but there are types of parallelism which static analysis will be unable to use. This parallelism is lost because of the inability of static analysis to make use of *dynamic data dependencies*. In a program where the control flow is determined at run time it is not possible, in general, to predict which branch in the program will be taken. A dynamic data dependency is said to occur when one section of code has a conditional dependency on another. A second form of dynamic data dependency arises due to variable aliasing. If the program uses dynamic binding of names to variables, e.g. when using references and pointers, it will not be possible to determine the data dependencies at compile time. Optimistic execution schemes are able to cope with dynamic data dependencies. This paper will discuss the use of optimistic execution schemes to overcome the problem of dynamic data dependencies.

In section 2 we will briefly describe discrete event simulation, and the techniques used in parallel discrete event simulation (PDES) so that the discussion in section 3 of using optimistic techniques for the execution of programs will be more readily understandable for those unfamiliar with PDES. In section 4 we give a design for an optimistically parallelising compiler. In section 5 we give a worked example, and we finish with conclusions in section 6.

2 Parallel Discrete Event Simulation

In discrete event simulation the "physical" system is modelled in terms of events, each of which corresponds to a state transition of an object in the physical system. Simulation events have a time–stamp which corresponds to the time at which the event would occur in the system being modelled. A sequential simulator proceeds by taking the event with the lowest time–stamp and simulating its effect: this may be to alter the state of an object being modelled, and may also create further events which will be scheduled at some future simulated time. The simulation moves forwards in simulated time by jumping from the time–stamp of one event to the next. This is in contrast to time–driven simulation methods where time moves forwards uniformly. The simulation is complete when there are no more events to simulate.

In parallel discrete event simulation [4] the physical system is modelled by a set of processes which correspond to interacting objects in the physical system. The interactions between the physical objects are modelled by the exchange of time–stamped event messages. Parallelism is achieved by placing these processes on the different nodes of a parallel computer. Each process has a logical clock which denotes the simulated time of that process. A process's logical clock will be increased to the time–stamp on each event message as it is processed. In this way the logical clock of each process will advance according to the time–stamps of events. The progress of the simulation is measured by the global virtual time (GVT), which is the minimum of the logical clocks (and time–stamps of messages in transit).

We must ensure that events are simulated in non–decreasing time–stamp order. To see why this is necessary, consider a process receiving a message which has an earlier time–stamp than its logical clock time. This means that the message represents an event which should have been executed earlier, and events have been executed past this which could have been affected by that event. This is known as a causality error, from the cause and effect principle: the fact that events in the future cannot affect events in the past. There are two approaches to ensuring causality is not violated in the simulation: the conservative and optimistic approaches.

Whereas conservative approaches avoid the possibility of any causality error ever occurring, optimistic approaches use a detection and recovery technique: causality errors are detected, and a roll–back mechanism is invoked to recover. Optimistic approaches are based on the virtual time paradigm [5]. A roll–back will be required when a causality violation is detected due to an event message arriving too late (this is known as a *straggler*). The roll–back must restore the state of the process in question to a point in time before the time–stamp of the straggler. After the roll–back, execution resumes from that point in time.

3 Using Optimistic Execution Schemes

To execute a program optimistically we have to transform the program so that it is compatible with the view of a system consisting of objects with a set of events which change the states of those objects. This view is reasonably close to the object–oriented programming model: we can view the invocation of object methods as events. If we take a program with a set of objects, and some method invocations, we can produce an optimistic version of this program by placing these objects on the nodes of a parallel computer. Objects will behave like Remote Procedure Call (RPC) servers: to invoke a method of an object we send the object a message requesting the invocation of the method. As this is an optimistic execution we do not wait for the invoked method to send the results back, but just continue executing. Parallelism is introduced into the system via the asynchronous dispatch of method invocation messages. The method will send the results back as messages when it has completed. Return values are sent to the object to which the return value is assigned, reference variables either affect the referenced variable during the execution of the method, or have their new values sent back to the referenced object when the method completes.

Server Objects We will associate a time–stamp with each method invocation such that by invoking the methods in non–decreasing order the same results will be achieved as for the original program. Each server object has a logical clock, and as each message is processed the logical clock is moved forward to the time–stamp of that message. If a message arrives with a time–stamp lower than the logical clock time, the server object rolls back its state to a time before the straggler message. In doing this it must cancel any messages it has sent to other server objects by sending out *anti–messages*. Then it reprocesses the events from the time to which it has rolled–back.

This scheme allows us to optimistically execute a transformed version of the original program. The optimistic execution is able to cope with dynamic data dependencies because it can assume the absence of data dependencies, and is able to recover when a causality violation occurs as a result of this.

Using optimistic techniques allows for more of the potential parallelism in sequential programs to be exploited. This is however not without cost: in order to execute code optimistically we need a roll–back mechanism; the maintenance of roll–back information, and the possibility of doing work which may later prove unnecessary must be balanced against the extra parallelism obtained.

4 Design of the Parallelising Compiler

In this section we will describe our design for a parallelising C ++ compiler and associated run–time system. The initial system is based on source–to–source transformations, we are using the sage++ transformation tool [2] to simplify this task. The transformations are directed by annotations added to the C ++ program. This is designed to facilitate the investigation of heuristics for adding the annotations automatically.

The compiler will need to effect the following changes to the sequential program:

- We need to be able roll–back the side effects of method invocations which we want to execute optimistically.
- We need to allocate time–stamps for method invocations. Some of these time–stamps will have to be allocated at run–time.
- We need some heuristics to decide on which C ++ objects would be suitable candidates for transforming into server objects.
- We need a placement scheme to allocate server objects to processors.

Roll–back We alter the class to a server object class so that it is able to roll–back. This transformation requires that we break server object methods into multiple smaller methods at the point where return values from RPC calls are received. The main program is itself considered a method of a dummy class for this transformation. This allows us to simplify the server object semantics so that we need only roll–back to the start of a method. The returned values are returned as RPC commands to the receiving server object. It will be necessary to roll–back the state of a control flow statement if the control construct outcome is determined by the state of a server object. Control flow constructs are transformed into server objects if this is necessary.

We use incremental state saving techniques to collect the information necessary for roll–back. Incremental state saving saves only the changes to the object's state, as they occur. The other commonly used state saving technique is periodic state saving where the whole state of the process is saved at regular intervals. We use the space–time memory model [3] as an optimisation to avoid unnecessary roll–backs caused by reading values from the past.

Time–Stamp Allocation Time–stamp allocation has to be done at run–time. To see why this is necessary consider the case where we time–stamp the method invocations in the body of a function. We can call the function multiple times, and the time–stamps will be different for each call. To perform time–stamp allocation for functions we pass a start time–stamp to the function, then we can allocate all of the time–stamps for the function offset by this time–stamp. Another reason we need to allocate time–stamps at run–time is that it is not always possible to determine at compile–time how many times a section of code will be invoked. Examples of this are recursive functions and loops whose termination is not fixed, but depends on a run–time evaluated expression.

Because of the unknown number of time–stamps we have to allocate for constructs such as loops with unknown bounds, our time–stamps must have similar properties to the rational numbers: it must be possible to allocate a sequence of time–stamps between any consecutive pair of already allocated time–stamps [1].

Object Selection Heuristics It is important that the objects selected as server objects have sufficiently large granularity methods, to overcome the overheads involved in message passing and roll–back information maintenance.

It is not possible, in general, to predict the execution time of a method, but one reasonable heuristic is to favour the higher level objects in the programs class hierarchy. As these will be written in terms of lower level objects it is reasonable to presume that they generally have longer execution times. Currently server objects are selected by annotation.

Object Placement In a distributed parallel computer which has a notion of locality it is preferable to place objects likely to communicate often on near processors. However the task of predicting communication patterns is itself not generally possible, so we must use heuristics to decide. Currently annotation based placement is used for object placement.

5 Example

We consider an example which has a dynamic dependency. Static analysis of this loop would reveal that the loop has a possible dependency and hence must be executed sequentially. An optimistic execution is able to parallelise the loop despite this possible dependency. The code is using a matrix class which has the usual operations, we will show the effects of an optimistic execution which is based on choosing all of the matrix objects as server objects, and executing their methods via asynchronous message passing. The `Matrix` class is transformed to become a server object `Server_Matrix`, and the invocations of its methods are transformed to RPCs. The flow of control taken by the if statement depends on the state of a server object and so is transformed into the `Server_If` server object. `det()` invokes the `eval1()` method of server object `if1`, sending that object its return value. `eval1()` uses the return value of `det()` to determine whether to invoke the `then1()` method which implements the then part of the if statement.

The original code:

```
class Matrix {
protected:
    // representation
public:
    Matrix();
    Matrix( const Matrix& );
    Matrix& operator=( const Matrix& );
    Matrix& inv();
    double det();
    // other operations
};

{
    Matrix w[ 100 ];
    ...
    for ( int i = 1; i < 100; i++ ) {
        if ( w[ i ].det() == 0 )
            w[ i ] = w[ i - 1 ];
        w[ i ].inv();
    }
}
```

The transformed code (simplified):

```
class Server_Matrix;
class Server_If1;

{
    Server_Matrix w[ 100 ];          // placed over processors
    ...
    for ( int i = 1; i < 100; i++ ) {
        Server_If1 if1;              // server for if statement
        w[ i ].det( if1, eval1 );    // RPC, result sent to if1
        w[ i ].inv();                // RPC call
    }
}

Server_If1::eval1( double res ) {    // receive result from det()
    if ( res == 0 )
        then1();                     // RPC to self to do body
}

Server_If1::then1() {
    w[ i ].operator=( w[ i - 1 ] );  // RPC to assign to w[ i ]
}
```

Say that the determinant `w[i].det()` is only equal to 0 for `w[3]`, then a possible optimistic execution of the above would be for a number of iterations to have been executed before the `if1` server processed its `then1` message for `w[3]`. The `then1` code would send an RPC to `w[3]` requesting an assignment, this would cause the server object `w[3]` to roll–back as it will have already processed the RPC request for `w[3].inv()`. The table below shows the sequence of events for server object `w[3]`, we have used time–stamps t in the range $i < t < i + 1$ for iteration i of the loop. When the `w[3]` has rolled–back to time 3.1, it resumes processing its event queue, executing the straggler event 3.2 and then re–executing event 3.3. We have managed to parallelise the loop where static analysis suggests we should execute it sequentially. The other server objects are not affected by `w[3]` rolling back.

logical clock	RPC message	message time–stamp
0	det()	3.1
3.1	inv()	3.3
3.3	op= (assign)	3.2 (CAUSALITY VIOLATION)
3.3	ROLL–BACK TO	3.1
3.1	op= (assign)	3.2
3.2	inv()	3.3

6 Conclusions

By assigning time–stamps to method invocations a conventional object–oriented program may be executed in parallel using an optimistic execution mechanism, with a guarantee that the same results are obtained as would be the case if the program was executed sequentially.

Further research needs to be carried out into the design of heuristics for code complexity measures, and for object placement. Also the inclusion of static data dependency analysis to avoid unnecessary roll–backs could improve performance in some cases.

References

1. Adam Back and Steve Turner. Time–stamp generation for optimistic parallel computing. In *Proceedings of the 28th Annual Simulation Symposium, Pheonix, AZ*. IEEE Press, April 1995.
2. Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An object–oriented toolkit and class library for building fortran and C ++ restructuring tools. *Object Oriented Numerics*, 1994.
3. Richard M Fujimoto. The virtual time machine. *SPAA (Symposium on Parallel Algorithms and Architectures)*, pages 199–208, 1989.
4. Richard M Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.
5. David R Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.