A PERFORMANCE ANALYSER AND PREDICTION TOOL FOR PARALLEL DISCRETE EVENT SIMULATION

ZOLTAN JUHASZ[1,3], STEPHEN TURNER[2],
KRISZTIAN KUNTNER[1] AND MIKLOS GERZSON[4]

[1]*Dept of Information Systems, University of Veszprem, Hungary*
[2]*School of Computer Engineering, Nanyang Technological University, Singapore*
[3]*Dept of Computer Science, University of Exeter, UK*
[4]*Dept of Automation, University of Veszprem, Hungary*
{juhasz, kuntner}@irt.vein.hu, ASSJTurner@ntu.edu.sg, gerzson@almos.vein.hu

Abstract: Developing parallel discrete event simulation programs is a difficult and time consuming process often ending with disappointment as the achieved performance is often lower than expected. Developers need performance prediction tools that are capable of providing information on the future performance of the parallel program prior to implementation. This paper describes a performance analyser tool developed to predict the performance of parallel discrete event simulation programs. Its unique feature is that it does not require the existence of the parallel program; it gathers the necessary program information from a trace file generated by a sequential simulation run. This enables one to determine, using a sequential simulation program, whether or not a parallel implementation is suitable, and if so, which approach is the best for the given simulation problem. The parallel version then only has to be developed if proved to be successful. The paper describes the analyser program, its architecture, functionality and use.

Keywords: parallel discrete event simulation, performance prediction, critical path, conservative protocol

1. INTRODUCTION

Simulation is a central tool in science and industry. The use of simulation enables us to investigate phenomena that otherwise would be impossible to study and understand. It helps us to learn more about the material world around us, design better machines, buildings, vehicles, and conduct experiments in order to aid decision-making. We can simulate systems as they evolve in time (e.g. a weather front or the phase change of a material) or as they change state (e.g. as a computer receives execution commands or clients in a shop are serviced). Since simulation programs are used to study the behaviour of complex systems in detail, they are computationally intensive and require long execution times.

Parallel computing technology has the potential to speed up the execution of large and complex sequential discrete event simulations. Despite many years of parallel simulation research, however, parallel discrete event simulation techniques have not yet gained widespread acceptance within the discrete event simulation community. This is mainly due to the facts that (*i*) developing parallel simulation programs is a difficult and time-consuming process requiring special expertise, and (*ii*) it is not guaranteed that the end product will deliver the required – improved – performance. The former requires the development and acceptance of very high-level (typically visual) parallel simulation programming environments (languages and tools), while the latter demands easy-to-use performance analysis and prediction tools.

Simulation users familiar with sequential simulation programs will only use parallel simulation techniques if there are suitable performance prediction tools that help them reason about the future performance (e.g. parallel execution time) of the parallel simulation program under development [Page and Nance, 1994; Page, 1999].

The achievable performance gain for any parallel simulation protocol depends on the particular simulation problem (application) as well as on many software and hardware related parameters. As a consequence, the prediction of parallel performance is a complex and difficult job. As there is neither a best parallel simulation protocol suitable for all situations, nor a universal recipe for achieving good performance, selecting the best parallel implementation is still frequently done by trial and error.

This paper describes the architecture and operation of a general-purpose performance prediction tool developed by the authors for parallel simulation. The tool is a post-mortem parallelism analyser that predicts parallel execution time for different simulation protocols, process-to-processor mappings and hardware parameters based on run-time information gathered during the execution of a sequential simulation program. The tool allows the user to explore the behaviour of different simulation protocols as well as the effects of different hardware parameters

prior to implementing the parallel simulation program, making it possible to decide very early in the development process which parallel solution (if any) is worth implementing. Using the tool, the developer can gain important insight into the execution of the simulation in question and, based on this information, select the best approach for its parallel implementation.

The paper is structured as follows. Section 2 provides a brief overview of sequential and parallel discrete event simulation. Section 3 describes the main trends in performance prediction applicable to parallel simulation and introduces the concept of critical path analysis. Section 4 discusses the design decisions behind our analyser. Section 5 gives details about the overall architecture of the program, and describes the detailed operation and implementation of each building module. Section 6 summarizes the major results. Section 7 draws some conclusions and briefly introduces further research issues.

2. PARALLEL DISCRETE EVENT SIMULATION

This section provides a brief overview of sequential and parallel discrete event simulation focusing on the main issues of parallel simulation execution. For more detailed discussion the interested reader is referred to [Fujimoto, 1990; Fujimoto, 2000; Wing, 92; Ferscha and Tripathy, 1994].

2.1. Sequential discrete event simulation

In discrete event simulations, we are interested in the evolution of systems, which is represented by changes of state that happen at distinct points of time as a result of characteristic events (e.g. an aircraft taking off). This behaviour is accurately reflected by the structure of sequential discrete event simulation models. As shown in Figure 1, the main components of the discrete event simulator architecture are (*i*) the event-processing engine, (*ii*) the global state vector, (*iii*) the simulation clock and (*iv*) a global event list that will contain the events that occur during simulation. Each event has an associated time stamp specifying when it is scheduled to occur, whose role is to help the system execute events in a correct sequence.

The execution of discrete event simulation programs follows a simple pattern. The simulation engine takes the event from the head of the event list, advances the simulation clock time to the time stamp of the event, then executes the piece of computation for the event and updates the state vector as required. Finally, if the execution of the event generated new (future) events, those are inserted in the event list. Note that events are held in non-decreasing time stamp order in the event list, hence both processing and inserting of events must follow time stamp order.

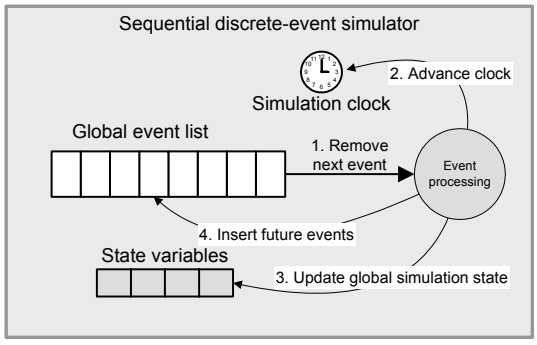

Figure 1. Architecture of a sequential discrete event simulation system.

2.2. Parallel discrete event simulation

In parallel discrete event simulation, we model the physical system as a collection of sequential simulations that represent different parts of the simulated system. These simulations are called logical processes or *LP*s with their own state; hence, a parallel simulation system is a distributed-state system. Just as in the sequential case, these *LP*s process and generate events. Consequently, each logical process has its own local event list, local simulation clock and local state vector. While it seems natural to map physical processes to logical ones and execute them in parallel, in reality there may be associated problems. It is possible that an *LP* generates events that influence other *LP*s, i.e. sets (sends) events to occur at another *LP* [Fujimoto, 1990]. This requires each *LP* to process not only its local events but also the ones generated by other processes, and in correct time stamp order. This necessitates a modification of the simulation architecture as shown in Figure 2.

The simulation system now contains multiple *LP*s and processors. Since processors can process events at different rates, it may easily happen that one processor is behind the others in simulated time; if an *LP* on this processor sends an event to any other processor, that *LP* will receive an event in its past, hence the processing of this event will violate the causality constraint imposed upon the simulation [Fujimoto, 1990].

As a result, the key problem in parallel simulation is how causality can be maintained, i.e. how processes are to be synchronised in order to process events in the right sequence. Based on the approach by which this is achieved, we can differentiate between two main types of synchronization (and so parallel simulation) mechanisms, the *conservative* and *optimistic* simulation protocols.

2.3. The conservative simulation protocol

The conservative simulation protocol works on the basis of avoiding causality errors and allowing only *safe* events to be processed. The difficulty in this

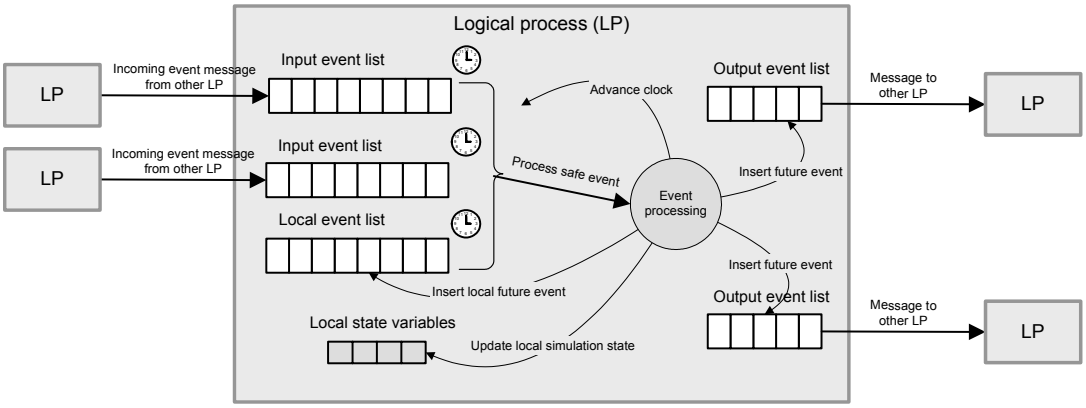

Figure 2. Architecture of a parallel discrete event simulation system showing the internal structure of a conservative parallel simulation logical process.

protocol lies in avoiding causality errors and deciding when an event is safe.

By definition, an event is safe to be processed if its execution will not lead to causality error. In practice, this means that the event is safe if it can be guaranteed that no event with earlier time stamp will ever be received on any input link or be present in the local event list. To achieve this, we need to impose a few more constraints on the system.

- Shared state – The system cannot have shared state, thus one process is not allowed to write into the state variables of other processes.
- Local causality – Logical processes must process events in non-decreasing time stamp order and generate events only in the future.
- Message delivery – The communication network must ensure that messages are delivered in the order in which they have been sent. This results in events being received in non-decreasing time stamp order if sent in non-decreasing order.
- Input waiting rule – Each event list has an associated clock set to the earliest time stamp in the list, or the time stamp of the last processed event if the list is empty. At each simulation step, the *LP* chooses the event with the earliest time stamp of all clock values unless the earliest time stamp is that of an empty list, in which case the process will block waiting for an input message on the given input link.
- Output waiting rule – A newly generated event can only be sent to another process if its time stamp is earlier than the time stamp of the next event plus the lookahead value, i.e. the next event cannot generate an event with an earlier time stamp than that of the just generated new event. The lookahead value is a measure of how far the process can look into its future and guarantee that no events will be processed during this time.

Under these constraints, it can be guaranteed that if *LP*s send and receive events in non-decreasing time stamp order, then the smallest time stamp event of all available events for a given *LP* is the safe event, and events are processed correctly. Note, while these contraints represent one (common) example of conservative protocol, there are other conservative protocols that relax some of these constraints.

2.3.1 Deadlock problem

This approach to selecting safe events works well if the input event lists are not empty. If a system has cycles of input lists, i.e. *LP*s receiving events from each other in a circle, and all these input lists become empty, no *LP* will be able to proceed as they will all block being unable to decide whether there is a safe event to process. Each LP will wait for an event to arrive into the empty list but none of them will ever become able to send one.

There are two fundamental approaches to dealing with this deadlock situation in conservative simulation (*i*) avoiding deadlock or (*ii*) detecting deadlock and recovering from it.

Deadlock avoidance is based on the null message mechanism. Special messages are sent after event processing to all processes for which an event was not generated to inform them about the earliest future timestamp an event can have leaving the sender process. On receiving the null message, processes will not perform computation; they will simply advance their simulation clock. The content of the null message is the current simulation time Ts_i plus the lookahead value γ, with $\gamma > 0$.

The other approach, deadlock detection and recovery, allows the simulation to deadlock but provides a mechanism to break such situations. A central controller is used to monitor processes: upon deadlock, it decides which process is safe to proceed, then notifies processors about which event is to be processed. This approach has substantial communication demand on the central manager and

usually requires specialized hardware supporting collective communication operations.

2.4. The optimistic simulation protocol

A fundamentally different approach to parallel simulation is based on the optimistic simulation protocol, also known as the Time Warp [Jefferson and Sowizral, 1985] algorithm.

The number one rule of the conservative approach has been to avoid causality errors by allowing processes to process only safe messages. In contrast, the optimistic approach will let *LP*s process events in any time stamp order but provide mechanisms to detect and recover from causality errors.

The following assumptions hold during optimistic simulation; an *LP* can send messages in any time stamp order, messages might not be delivered in the order they have been sent, and the number of *LP*s can change dynamically without the need to specify which *LP* is connected to which other one. The internal architecture of an optimistic *LP* is somewhat different from that of the conservative *LP*. In an optimistic *LP*, there is one event set including both internal and external events. This event set will include both unprocessed and processed events, whereas in the conservative *LP* processed events are removed from the event list. Keeping the processed ones in the list is necessary, as the *LP* might need them when a causality error occurs.

The two major issues in an optimistic simulation system are the detection of a causality error, and execution of rollback, i.e. restoring a previous simulation state.

2.4.1 Detection of causality error

Since events can be sent in any order, it is expected that an *LP* may receive events with earlier time stamps than the ones being processed. Such an earlier time stamp event has not been processed at its simulation time, therefore the *LP* must roll back to a simulation time preceding this event undoing the already simulated events, then restart again simulating all events in the correct order. The consequence of the rollback is twofold. During the processing of an event, an *LP* can can modify state variables and send messages to other processes (schedule new events). Thus, to undo processing, a previous state of the *LP* needs to be restored and the effects of sending messages to other *LP*s need to be cancelled.

2.4.2 Simulation rollback

Restoring previous states requires saving states during the simulation. There are two fundamental approaches to saving state variables. *Copy state saving* makes a copy of all the state variables prior to processing each event. Hence, a snapshot of the system is created for each event, which can easily restore the state for any previous simulation time. The other approach, *incremental state saving*, only saves the modified state variables. A log is created that lists all state variables along with their new values and modification dates. Rolling back to a previous state in this approach requires the system to work backwards with the modification log and restore the state event by event in reverse order.

Cancelling sent events is a more complicated problem, since it is possible that a sent message has already been processed by another process and perhaps, as a consequence, already sent other messages with new events to other processes. The effect of one message can thus spread throughout the entire simulation system. We need ways that guarantee the correct cancellation of all these messages. In optimistic simulation, “anti-messages” are used for event cancellation. The anti-message is a time stamped message requesting the cancellation of an earlier sent message. For each new message, an anti-message is created and stored in the local process. Whenever a message has to be cancelled, the stored anti-message is sent to the relevant processes. By receiving the anti-message, the process will roll back to the latest swaved state preceding the time stamp of the anti-message and re-start the simulation (and send further anti-messages if necessary).

There are several advanced issues related to optimistic simulation not covered in this paper. These relate to how and when state saving is performed, when to send and evaluate anti-messages, how to reclaim memory used for storing previous states, how to compute the Global Virtual Time in the simulation, etc. The reader is referred to [Fujimoto, 2000; Ferscha and Tripathy, 1994] for further details and information.

3. PERFORMANCE PREDICTION OF PARALLEL SIMULATION

The ultimate goal of parallel discrete event simulation is to provide improved performance to the simulation community. This goal is difficult to achieve. It is evident from the previous section that a parallel simulation program is a complex dynamic system. The performance of the final parallel simulation program depends on many application and system level parameters, such as the logical structure of the simulation, the chosen simulation protocol, the number of processors, the interconnection system and their hardware parameters; these are all critical to performance. Since the development of parallel simulation systems requires time and special expertise it is important to be able to make design decisions that will lead to a system of acceptable performance.

In parallel simulation, there is no recipe for how to achieve optimum performance, as performance is the result of the complex interaction of many different parameters. For some systems, conservative simulation is a better choice; for some others, the optimistic approach is more suited [Fujimoto, 1990]. Variations

of both protocols exist that improve performance for specific application contexts.

Due to its complex nature, parallel simulation will only be used if development is based not solely on trial-and-error, but assisted by user-friendly performance prediction tools that enable the developer to reason about the future performance of the parallel simulation program under development [Page and Nance, 1994; Page, 1999].

An effective prediction technique provides insight into the behaviour of the parallel algorithm, reflects the impact of source code variation on program performance, and allows the programmer to predict the performance of hypothetical systems and algorithms. A good method should be accurate, simple and quick to use in order to be incorporated into the design phase of the development. Simulation prediction tools allow one to explore the behaviour of different simulation protocols as well as the effects of different hardware parameters prior to implementing the parallel simulation program, making it possible to decide very early in the development process which parallel simulation protocol (if any) is worth implementing.

Research on parallel simulation performance prediction can be grouped into two main schools, the *analytical* and *simulation/trace based* approaches.

3.1. Analytical performance prediction

Analytical methods use mathematical models of the architecture and the execution of the algorithm, and express the execution time in closed-form expressions [Clement and Quinn, 1993; Crovella and LeBlanc, 1993; Grama *et al*, 1993]. Because performance is estimated through a set of equations, the analytical method provides a powerful, flexible and elegant way to study and explore performance even when a system is not yet implemented or it is impractical to build. Performance predictions are based on parameters obtainable on a minimal, few-processor system. They examine the effect of system parameter changes and they help to optimise a system for a given choice of parameters. They identify regions in the parameter space that provide good or poor performance, and compare different algorithms and systems. As a result, analytical methods are, in general, best suited for performance prediction.

Due to the dynamic nature of parallel simulation systems, however, deterministic analytical models cannot be used to predict the performance of discrete-event simulation programs. Hence, simulation performance evaluation and prediction are based on stochastic models. Lubachevsky was the first who studied the effect of synchronisation in a synchronous conservative simulation system, and developed a worst-case performance model [Lubachevsky, 1988]. Later, Nicol [Nicol, 1993] improved upon his results and created an average case model. The effect of lookahead on conservative simulation has been studied by several researchers [Lin and Lazowska, 1989; Wagner and Lazowska, 1989; Fujimoto, 1988]. Other critical performance issues are null-message generation policy, mapping and communication (hardware) parameters [Nicol, 1993; Felderman and Kleinrock, 1992].

The analysis of optimistic methods is even more complicated. Hence, only small systems have been investigated. Performance models for two-processor systems have been set up by [Lavenberg *et al*, 1983; Mitra and Mitrani, 1984]. An approximate model for larger systems has been constructed by [Gupta *et al*, 1991].

The main disadvantage of using analytical prediction approaches for simulation is that their construction is extremely demanding and the resulting model is very difficult to validate.

A fundamentally different approach to prediction is based on actual execution trace information and critical path analysis.

3.2. Trace-based performance prediction

The underlying concept behind critical path prediction [Srinivasan and Reynolds, 1993] is that once the parallel execution graph of a program is known, the critical path gives the shortest possible execution time (at least for a conservative simulation as it is possible to execute a simulation in less than critical time using optimistic techniques). The trace information necessary for building the execution graph can be generated by either (*i*) parallel or (*ii*) sequential simulation program execution. Obtaining trace from a parallel simulation run has the advantage of perfect accuracy but, unfortunately, it requires the implementation of the parallel simulation program and corresponding hardware; hence this approach cannot be used for prediction a priori to implementation.

One would rather work from a sequential simulation program and implement a parallel one only if proved to be worthwhile. Using a straightforward sequential trace for prediction is not adequate, however, as the critical path in this case does not include overheads due, for instance, to event synchronisation that depend on the simulation protocol used.

The problems of predicting the execution time of a parallel simulation program from a sequential trace are best illustrated using the following simple example. Assume a simulation system consisting of five processes P_1, P_2, … P_5 as shown in Figure 3. The arrows show communication paths, or links, for the transmission of events generated for other processes. Assume also that each process executes and schedules events.

An example event precedence graph, describing the inter-dependencies among events is shown in Figure 4.

Here process P_1 generates and schedules new events for itself (e_5, e_7) and for processes P_2 (e_2) and P_3 (e_6, e_8). P_2 and P_3 in turn generate and schedule events for processes P_4 (e_3, e_{10}) and P_5 (e_{11},e_{13}). Finally, P_4 generates events (e_4, e_{15}) for P_5. We further assume that the timestamp of each event is equal to the value of the event's index (for instance, e_2 occurs at time 2) and that the execution of each event takes 3 time units.

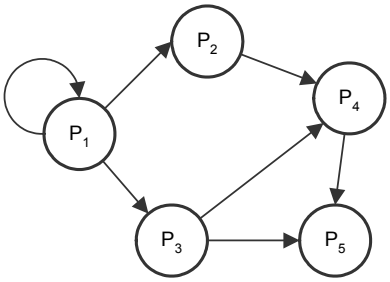

Figure 3. A 5-process sample simulation system

The sequential execution of this simulation ensures the correct execution order of events, resulting in the sequence of e_1,e_2,…e_{15} with an overall execution time, T_{seq}, of 36 time units.

Since the event precedence graph is non-cyclic, there is a maximum weighted path (a critical path) that represents the minimum parallel execution time. This representation ignores the effects of parallel simulation protocols; therefore the critical path execution time (when assuming zero communication delays) gives an absolute lower bound on the parallel execution time. In the example, the critical path is e_1, e_2, e_3, e_4, e_{11}, e_{13}, e_{15} with an execution time of 21 time units. This results in a maximum achievable speedup S_p of 36/21 = 1.71, regardless of the number of processors.

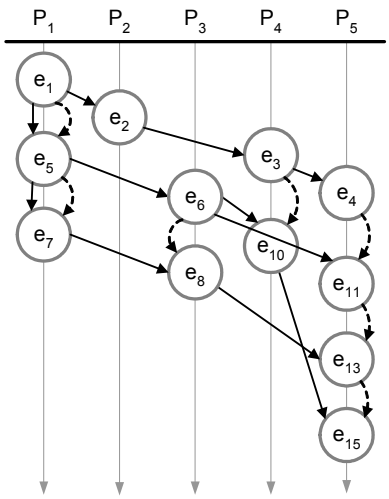

Figure 4. The event precedence graph for the system: solid line arrows represent event scheduling dependence, whereas dashed ones mark event causality constraints.

If this simulation is executed correctly using for instance a conservative simulation protocol, there will be inevitable delays (illustrated in Figure 5.) as processes are only allowed to proceed with the simulation if the corresponding events are safe to process, in other words their execution will not introduce causality errors.

Figure 5 illustrates the timing of parallel execution when using a conservative protocol with null-messages. In our example, a lookahead value of 3 time units is assumed for all processes. Based on the rules just described, the conservative parallel execution introduces delays. Process P_4 needs to wait for a message on each of its input links in order to be able to proceed (input waiting rule), whereas event message e_{10} cannot be sent out at the end of e_6 because at that point it cannot be decided whether the next event, e_8, would produce an event with an earlier time stamp than 10. Process P_3 can guarantee the occurrence of no events only up to time stamp 9. The small grey boxes represent buffers where messages wait until they are considered to be safe for sending and processing.

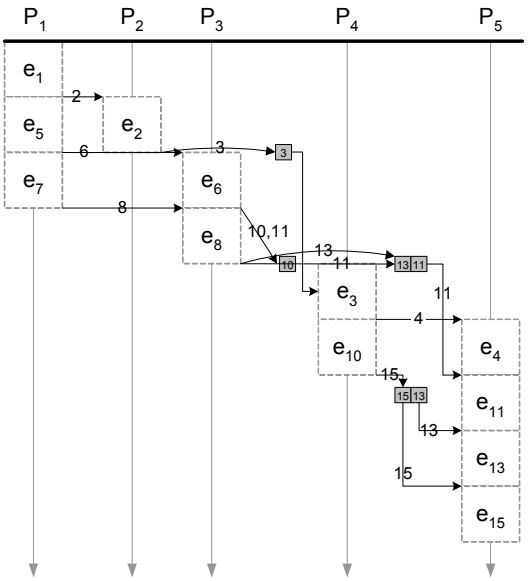

Figure 5. Sample simulation execution according to the conservative simulation protocol.

The resulting execution time has increased to 27 units (S_p = 1.33). In real systems, other delays may be observed. For example, communication cost normally cannot be neglected. Consequently, execution times will be longer, and speedups smaller then the ones indicated by the critical path analysis. The effects of these parameters can be crucial, and because they are difficult to analyse, the need for performance prediction tools is inevitable.

4. THE PERFORMANCE ANALYSER

A possible approach to create a parallelism analyser based on sequential execution is to build the analyser

into a sequential simulator, which then performs protocol-based synchronisation analysis as the sequential simulation is run. Two such systems are described in [Lim *et al*, 1999; Wong *et al* 1995]. The former is used for the Chandry-Misra conservative protocol, while the latter is for the synchronous conservative protocol. An alternative to this method is the a posteriori analysis that uses a sequential trace for the same type of analysis.

Both approaches have advantages and disadvantages. The integrated approach (analyser within the simulator) does not need to store trace information (normally in a file) that can be very large for extended simulation runs. On the other hand, sequential simulation must be executed over and over again when the effect of a new parameter value is to be investigated, and the simulation program needs to be modified if new protocols and/or functionality are to be added in.

The posteriori approach (stand-alone analyser) requires a trace file but has the advantage that the simulation needs to be run only once for the prediction (exploration of the parameter space) to be performed at different locations, in many cases faster, and in a more intuitive way – making it possible to change the simulation protocol, process-to-processor mapping, communication, etc. parameters on the fly. This approach has been followed in the analyser program developed by the authors. Such an analyser studies the effects of changes in architectural parameters (number of processors, speed of processors, interconnection topology and communication system parameters – distance, startup time, per word transfer time–, process-processor mapping (e.g. manual, random, bin-packing, min-comm), and simulation protocol (conservative with different deadlock avoidance approaches, null message generation policies, lookahead size; optimistic protocol with different state saving and message cancellation policies). In addition, this approach does not require the use of any particular simulation or programming language. Existing sequential simulation programs can be used as long as they generate the required trace information, which normally requires only minimal program modification.

The analyser tool developed for this research helps developers answer the following types of questions. (*i*) What is the shortest possible execution time (or largest speedup) that a parallel implementation could provide? (*ii*) Which simulation protocol will give the best results for my particular application? (*iii*) How will the speedup scale if we increase the number of processors or the size of the problem? (*iv*) Which parallel architecture is the most suitable for our implementation?

There are several situations in which the tool can be useful. Given an existing sequential simulation program, it is first possible to determine whether a parallel implementation would sufficiently reduce execution time. Focusing on a particular parallel computer as optimization target, the most suitable parallel simulation protocol for a given performance objective by adjusting the protocols and mapping strategies. If the goal is to find the most suitable parallel machine, the performance of the parallel simulation can be investigated on "virtual" parallel machines by changing the value of the characteristic hardware parameters. Finally, the tool can be used to predict the performance parameters for different numbers of processors, providing important scalability information.

The ability to quickly explore the effect of different hardware, software and architectural parameters will help developers in gaining insight into the behaviour and sensitivity of the parallel simulation program under development.

5. ARCHITECTURE OF THE ANALYSER

The architecture of the analyser program follows the natural grouping of its base functionality. It is a modular, object-oriented architecture that reflects the underlying responsibilities of the program and facilitates future system extension. As shown in Figure 6, the main modules are Trace, Analyser, Architecture, Mapping and Performance.

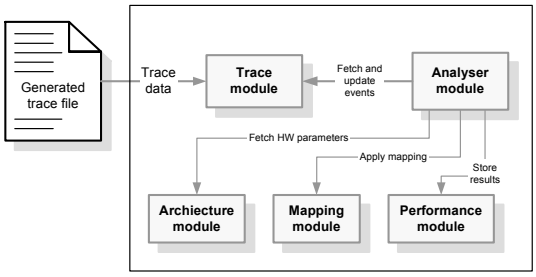

Figure 6. The high-level structure of the performance analyser and prediction program.

The Trace module is the repository of the simulation events that are read in from the sequential trace file. As events are read, the Analyser module processes them to calculate exact event start times and message transmission details, based on the particular simulation protocol selected by the user. During processing the Analyser module interacts with the following modules:

- Trace module to fetch and update events,
- Architecture module to access the properties of the target parallel architecture,
- Mapping module to determine how processes should be allocated to processors,
- Performance module to record the results of the analysis and predictions for the parallel execution.

In the remaining of this section, the operation and the internal architecture of each module is described in detail.

5.1. The Trace module

The trace module stores events and their inter-dependencies. The analysis starts with raw events that will be transformed as the analysis is being carried out. Since the input to the entire process is the sequential simulation trace, the contents and the structure of this file is described first.

5.1.1 The trace file format

A general sequential simulation trace, not specifically designed for parallel execution prediction, is simply a list of events executed in timestamp sequence. As the event precedence and execution graphs are needed in order to calculate the critical path, an extended trace file is required that contains additional information about the events.

For each current simulated event, all the necessary details including information about its future generated events are recorded. Undoubtedly, some pieces of information, such as the execution length in simulated and real clock time cannot be known for future events, as these are scheduled for later execution. These details are stored when they are executed as current events.

The structure of the trace file is given below in Bacchus-Naur Form notation. To assist users in creating trace files, a small routine library has been developed providing special trace functions that can easily be inserted into any existing sequential simulation program.

```
<trace> ::= <numberof-processes> <EOL>
            <tracelines>
<tracelines> ::= <current-event>
            <future-event-list>
<current-event> ::= <process-ID> ",” <event-
            ID> ",” <source-event-ID> ",”
            <timestamp> ",” <simulated-
            execution-time> ",” <physical-
            execution-time> ",”
<event-ID> ::= [eos-tag] <event-count>
            "@”<source-process-ID>
<eos-tag> ::= "$”
<future-event-list> ::= <future-event>
            <EOL> | <future-event>
            <future-event-list>
<future-event> ::= <event-ID> ",”
            <destination-process-ID> ",”
            <timestamp> ",”
            <message_size> ",”
```

An example trace line is shown next illustrating process P_2 executing an event $id = 6@1$ generated by process P_1 with timestamp $ts = 7$, simulated execution time $T_{sim} = 5$, real execution time $T_{exec} = 76$ μsec, and generating event $id = 3@2$ for process P_3 at time $ts = 12$ transmitted in a 24-byte long message:

```
2, 6@1, 1, 7.00, 5.00, 0.000076, 3@2, 3, 12.00, 24,
```

5.1.2 Data structures

The internal data structure of the module follows that of the trace file. A trace object is created that holds instances of the Event class in an event set that is the central repository of events. Current and future events are added as the trace file is read. The trace object also holds a pointer structure that matches the trace file structure: a list of trace line objects, each having a current event pointer as well as a pointer list accessing future events. This structure is depicted in Figure 7.

Since the program has been implemented in Microsoft Visual C++, the Microsoft collection classes are used. However, it is straightforward to modify the system for the use of standard template library collection classes. The trace object additionally stores a process topology object, the role of which is to specify the interconnection structure of simulation processes, which should be known for the purposes of analyses based on conservative protocols.

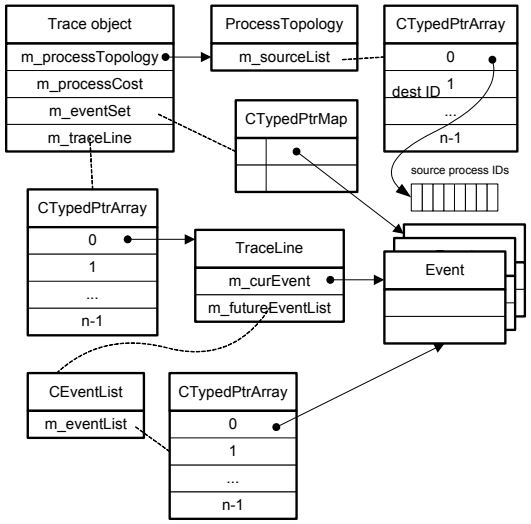

Figure 7. The event trace data structure.

As a trace line is processed, the program creates a new event object for the current event with the required parameter settings, and the additional events that it will generate in the future. New objects are added to the event set with the corresponding pointer structures being updated. If the current event already exists in the event set (stored previously as a future event), it is updated with the now know execution times.

What is not known at the end of the trace file processing is the actual real clock starting times for each event execution. It is the role of the Analyser module to determine and set these values.

5.2. The Analyser module

The Analyser module is the central component of the system. Its main responsibility is to emulate the parallel execution of the simulation based on a user-selected simulation protocol. During the analysis, the

module retrieves the simulation events stored in the trace module, and determines the exact start time of each event as well as the message transmission parameters (send time, arrival, processing time) based on event inter-dependencies, hardware parameters and protocol execution rules.

5.3. Protocols

Along with hardware parameters and mapping algorithms, users can choose from a wide variety of conservative and optimistic parallel simulation protocols. Unfortunately, there is no recipe for success when choosing one. Finding the best protocol is a trial and error process, making it ever more important to be able to perform these analyses without writing a parallel implementation.

To make the system easily extensible, a general analyser interface (implemented as an abstract class with pure virtual functions) was created to define the operations for the analysis process. The starting point for analysis is the *CAnalyserBase::analyse()* function. Each simulation protocol in the system implements this interface and provide an appropriate implementation for protocol analysis in the *analyse()* function. A sample hierarchy of conservative and optimistic protocols is shown in Figure 8.

The implemented analyser algorithm is developed for the Chandry-Misra conservative simulation protocol with null messages generated after the occurrence of each event. Our algorithm is based on [Wong *et al*, 1995] and has been modified to become suitable for posteriori analysis. The algorithm can be altered to deal with different null-message generation policies and deadlock avoidance algorithms.

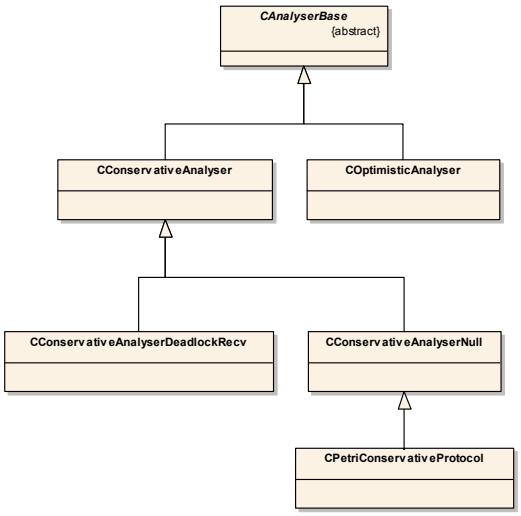

Figure 8. The inheritance hierarchy of the simulation protocols.

The first analysis of any trace file is performed with the default settings of (*i*) conservative simulation protocol, (*ii*) one process per processor ($P = N$, where P is the number of processors, N is the number of processes in the system), and (*iii*) zero communication costs. This results in an ideal emulated parallel execution that in turn specifies the best achievable parallel performance for a conservative simulation. Subsequent analyses can be performed with different protocols, realistic communication parameters and using $P < N$ processors.

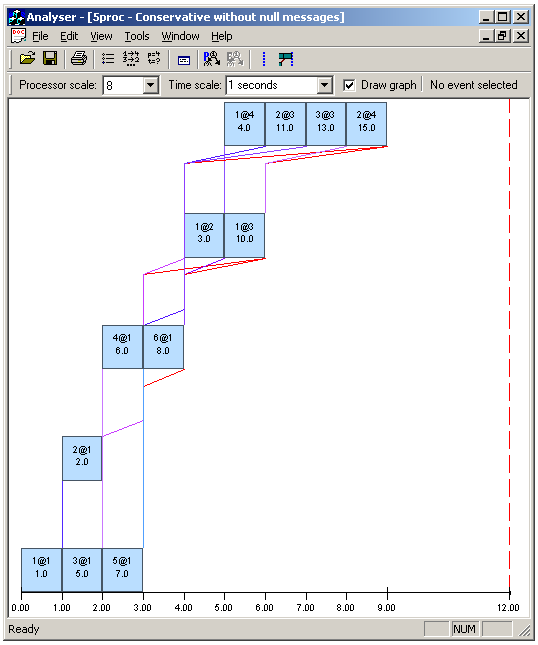

Figure 9. Predicted parallel execution of the sample simulation using conservative protocol without null messages, with zero communication cost and $p = 5$ processors.

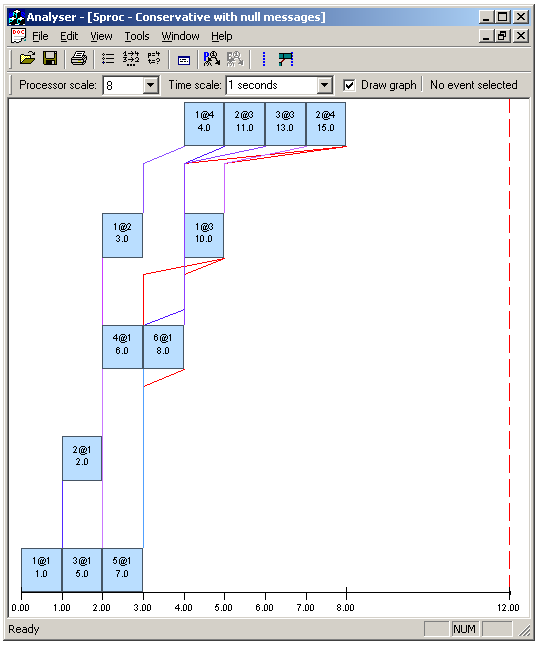

Figure 10. Predicted parallel execution of the sample simulation using conservative protocol with null messages.

Figures 9 and 10 illustrate the effect of protocols. The conservative protocol used in Figure 9 does not use null messages (in practice, this protocol would cause deadlock); therefore, process P_4 cannot be simulated until events e_3 and e_{10} are received from processes P_2 and P_3. Since e_{10} is delayed due to the output waiting rule, P_4 starts at execution time t = 4.

If null messages are used (Figure 10), every process will generate a null message to its downstream processes if a regular event message has not been sent. Hence, process P_3 generates one at the end of processing e_6 resulting in P_4 starting execution at time t = 3.

5.4. The Architecture module

The Architecture module abstracts out and represents various types of target parallel architecture that can potentially execute the simulation under study. Note that since a sequential simulation is used as input, any architecture and its effect on performance can be investigated prior to implementation.

If the initial execution time and speedup results indicate that a parallel implementation is worthwhile, the user can investigate the effects of changes in hardware, architectural parameters, as well as the consequences of using different mapping approaches and simulation protocols.

5.4.1 Architecture descriptors

In the current system, homogeneous, dedicated parallel architectures can be modeled, consisting of identical processors connected by either a bus or point-to-point message passing interconnection. The major architectural characteristics are the types of processors and the available interconnections.

Processors are characterised by a single parameter only, a relative speed factor (as compared to the speed of the processor used to generate the sequential trace). This constitutes a simple way to examine the effects of execution speed on performance. This feature establishes whether performance is communication or computation bound, and anticipates the effect of future improvements in microprocessor technology.

The interconnection system is represented by the Topology class that is dynamically substituted by derived classes due to the polymorphic property of object-oriented systems. All topologies must implement this class as shown in Figure 10. representing the class hierarchy of the implemented interconnections. The primary aim of the interconnect classes is to return the distance d_{ij} between processors P_i and P_j in a topology-dependent manner. Changing from a fully connected topology to some other interconnect may introduce – besides the possible increase in distance – the sharing of the communication device. In many cases this leads to contention (on the bus or link), which in turn introduces further communication delays.

In our analyser, we use the cost model for message communication of $T_{comm} = T_s + kT_w + rT_h$, where T_s is the message startup time, T_w is the per-word transfer time, T_h is the per-hop time, k is the number of words in the message and r is the number of routers the message has to pass through (if the interconnection is a switch-based one). Message startup time is often the most important parameter in a parallel system. Its value ranges approximately from 2 μsec (fastest dedicated parallel computers) to a few hundreds of microseconds. Since the per-word transmission and per-hop times are normally fractions of microseconds, startup time can easily dominate message transfer time.

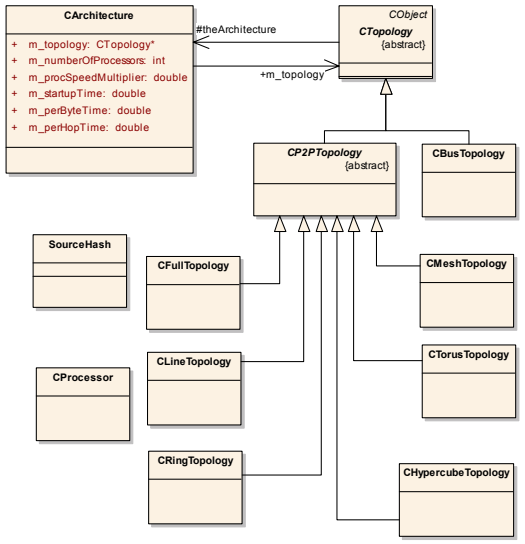

Figure 10. Static class structure of the architecture module.

By changing the value of the startup parameter, the user can quickly identify how much additional delay will be introduced by the target communication hardware, and decide on the range of acceptable values for T_s. Should the user think that the values of T_w and/or T_h are important, they can be adjusted similarly.

5.4.2 The effect of architectural parameters

The analyser can be controlled through a graphical user interface. The architectural parameters can be modified in the hardware parameter dialog window, shown in Figure 11. By default, this window shows the number of processors equal to the number of processes, using a fully connected interconnection topology with zero communication cost.

The user has the option to change the relative speed of the processors, the interconnection system and its topology, the message startup time, per-word transfer time, and per-hop time. Figure 12 shows an example where the message transmission parameters are considerably increased. For small simulations the parameter modification and the subsequent analysis can be performed quasi real-time, hence by moving the slider controls, the change in performance is computed

and displayed instantaneously in the dialog box. This approach can be used for fast exploration of the parameter space. For large simulation traces this instant analysis can be disabled, and be performed only by explicitly pressing the button 'Re-Compute'.

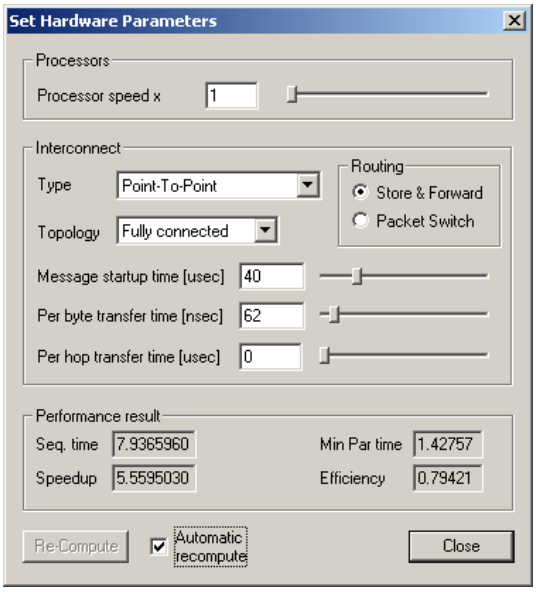

Figure 11. Dialog window for changing architectural and hardware parameters.

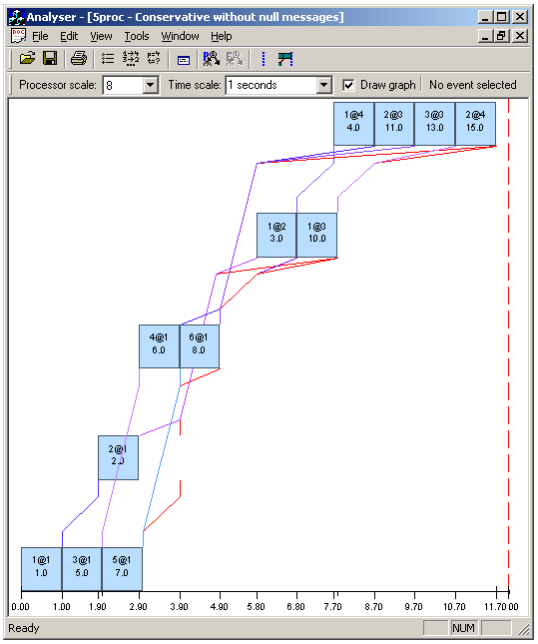

Figure 12. Predicted parallel execution of the sample simulation using conservative protocol with non-zero communication cost.

5.5. The Mapping module

The last feature of the analyser to be discussed is the process-to-processor mapping. In many cases, there may not be as many processors available (very large simulations) as simulation processes or alternatively the same performance could be achieved using fewer processors. This latter aspect is important as it increases the efficiency and utilisation of the parallel system, and consequently can reduce cost. The specification of the number of processors is closely related to the problem of mapping processes to processors. With fewer processors than processes, processors are responsible for executing several processes. The way that processes are mapped to processors can have a significant effect on system performance.

Figure 9 shows for example, that there is no parallelism between processes P_1 and P_5, and between P_3 and P_4. In relation to one another, they execute their events in a sequential manner. Thus, without degrading performance, we could map P_1 and P_5 onto the same processor. The same is true for P_3 and P_4. This way, the same problem can be solved within the same time using only three processors. P_1 and P_2 may be mapped onto the same processor. This would only marginally reduce performance but would only require a system of 2 processors. The automated mapping of processes is, unfortunately, a difficult task.

The problem of mapping has been central to parallel computing. Bokhari showed [Bokhari, 1981a] that the general mapping problem is *NP*-hard. Polynomial solutions exist only for a small set of specific cases, such as mapping arbitrary programs onto a two-processor system [Stone, 1977] and mapping chain [Towsley, 1987] or tree-structured [Bokhari, 1981b] parallel programs to processor networks.

Approaches to the solution of the general mapping problem are based on: graph algorithms [Bokhari, 1981b], heuristics [Chen, 1995; Woodside, 1993; Nicol and Mao, 1995; Nandy, 1993], combinatorial optimisation methods (e.g. branch-and-bound, dynamic programming) [Stone, 1977], AI methods (genetic algorithms, simulated annealing), and neural networks (self-organising networks) [Heiss, 1996; Shen, 1992]. The majority of heuristic methods that have been proposed are based on either bin-packing algorithms or node merging methods. Probably the most widely known mapping algorithm is the bin-packing algorithm, called MULTIFIT, proposed by Coffman [Coffman et al., 1978]. Woodside and Monforton [Woodside and Monforton, 1993] proposed a modified version of this algorithm. Their MULTIFIT-COM algorithm can take inter-task communication costs into account for bus-based parallel systems.

The authors developed an extensible class structure for the mapping module (Figure 13). A mapping base class *CMappingBase* abstracts out the functionality of the mapping module, and every mapping algorithm implements this interface (inherits the base class). The

base class is associated to the process map: a lookup table holding process-processor mappings.

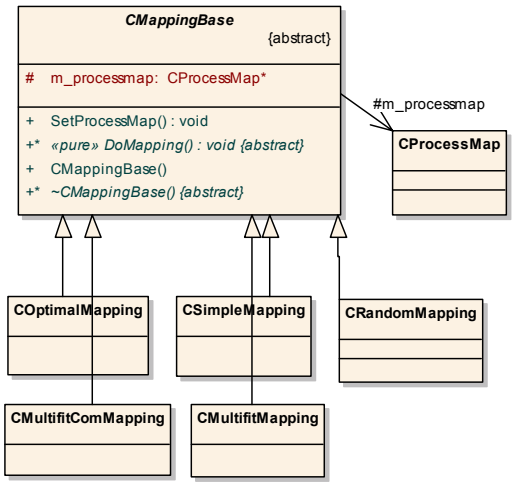

Figure 13. Static class structure of the mapping module.

The user can select from any of the available mapping algorithms (the default is one-to-one mapping) and the chosen algorithm will be used whenever the number of processors is changed during the analysis. This way, the user may explore the performance of a particular mapping as the different hardware parameters are changed. It is important to note that changing the number of processes changes the overall communication cost as well, as communication links may become intra or inter-processor ones. Intra-processor links are usually implemented in memory, hence are faster then communication over the physical external network. A change in the mapping can also alter the contention pattern on a shared communication resource. Figure 14 and 15 show examples for the effect of process-to-processor mapping.

In the case of Figure 14, a modulo mapping algorithm is used (the default algorithm in Parsec), defined as $ID_{target\_processor} = ID_{process} \bmod P$, where P is the number of processors in the target architecture.

On the other hand, in the case of Figure 15, an exhaustive search-based partitioning algorithm has been used that results in ideal load balancing on the target processors. It can be seen that balancing computation and/or minimising communication on its own is not an adequate optimisation strategy. Due to event inter-dependencies and protocol synchronisation constraints, a sub-optimal mapping – in terms of load balancing – can provide better performance.

Another relevant aspect of the mapping process is the way processes mapped onto the same processor are treated during event processing. The analyser provides the option to treat these processes as separate entities that are concurrently executed on the same processor, in a time-sliced manner. In this case (assuming a conservative protocol), if at least one process on a given processor has a safe event, the processor will work.

The other option is to treat mapped processes as a single process by merging the original processes. Then, during conservative simulation, the new process will wait for events from its new output links. This approach can increase the computational granularity of a parallel simulation to achieve better performance.

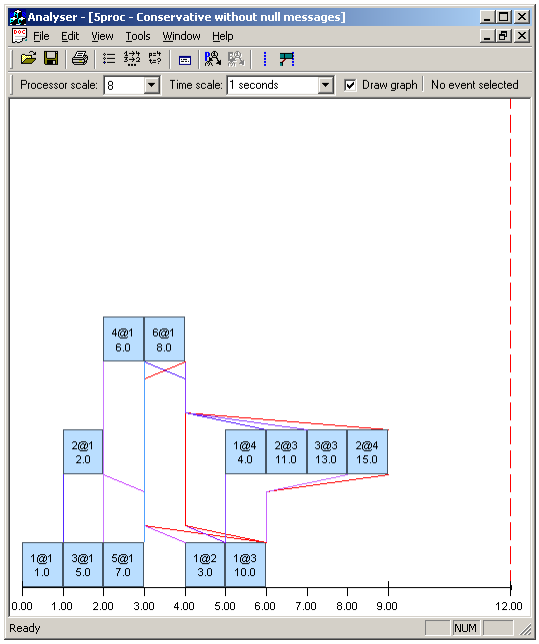

Figure 14. Predicted parallel execution of the sample simulation using conservative protocol with zero communication cost 3 processors. Process 4 is mapped to processor 1, process 5 is mapped to processor 2.

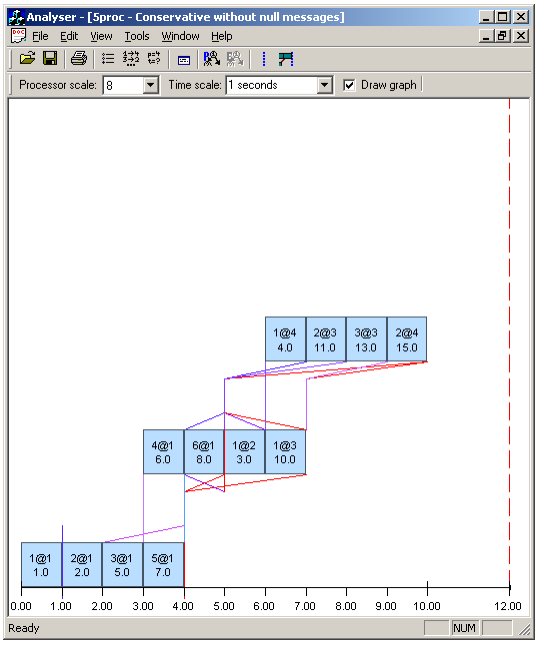

Figure 15. Execution of the sample simulation on three processors using the ideal partitioning algorithm, illustrating the influence of event inter-dependencies on mapping.

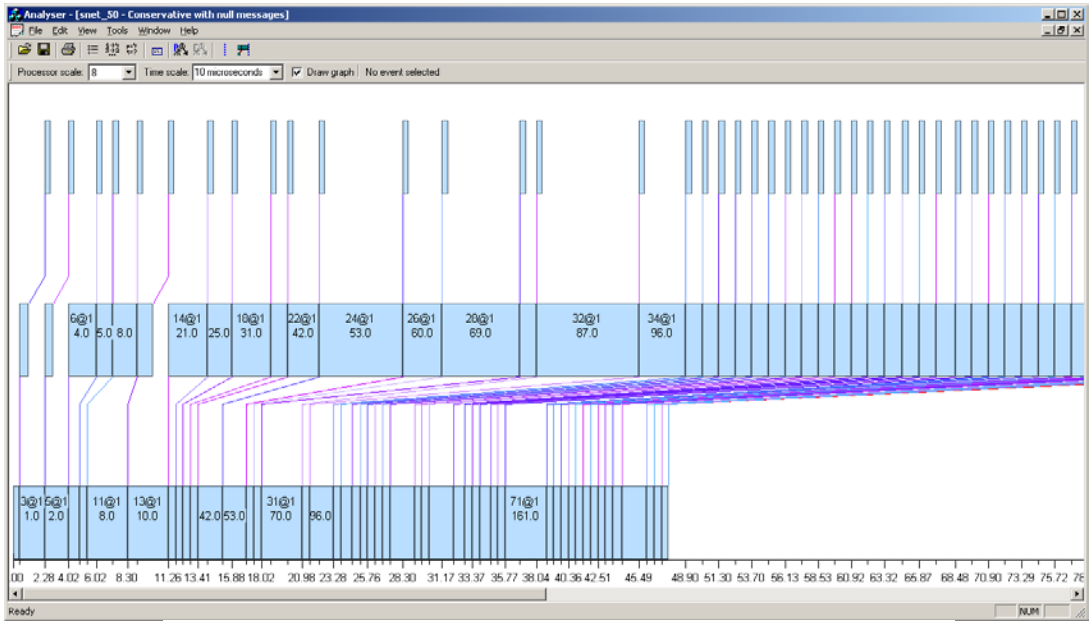

Figure 16. Screenshot of the graphical user interface showing a portion of the predicted parallel trace of a real simulation program.

5.6. The graphical user iterface

The analyser program has a graphical front end for visualising the predicted parallel execution based on the input sequential execution trace, as well as setting architectural and mapping parameters, and selecting the applied simulation protocol.

The predicted trace can be viewed at different scales; the processor and time scale of the display window may be modified for at-a-glance overview or detailed trace inspection. A sample screenshot is shown in Figure 16.

6. RESULTS

The current version of the analyser provides users with a choice of conservative simulation protocols. It also enables the study of the effects of architectural changes as well as various process-to-processor mapping methods.

One of most important goals of this research was to provide fast analysis facilities for program designers. The execution time of the analysis for the current non-optimised version was compared to that of sequential simulation runs and the corresponding per event analysis time was found to be in the range of 15-50 μsec. If the simulation under study has a higher granularity, the analyser provides shorter analysis cycles. Table 1 lists execution and analysis times obtained for a sample 3-process simulation program. For a 7-process, larger granularity simulation (app. 2200 μsec average event processing time), the analyser was 50-60 times faster than the original, full speed simulation.

Table 1. Simulation and analysis execution times for different numbers of events of a 3-process simulation.

	Time [msec]		
Number of events	Sequential simulation program (Parsec)	Seq. simulation program (Parsec) generating trace	Analyser (analysis time only)
3000	100	360	42.12
4500	250	431	64.92
6000	250	461	84.84
9000	260	561	126.83
12000	300	661	167.49
30000	301	1252	421.47
300000	580	5297	4230.00

Accuracy is another important issue for the purposes of the analysis and performance prediction. The analyser validation has not yet been completed but current results are encouraging. Figure 17 presents the actual and predicted results for a 7-process parallel simulation. As shown in both Figure 17 and Table 2, the predicted speedup is in line with the measured one (prediction error is less than 10 %) for systems including up to 5 processors, for larger numbers of processors (larger than 5) the error becomes 50-60%. Further analysis is required to identify the reason for this error increase.

Several parallel execution details cannot be taken into account during analysis and prediction, for instance context switching time, internal housekeeping and event queue management of the parallel simulation engine, etc. as the the model relies on event processing time only. Further analysis is also needed to identify the overheads of sequential executions, in other words

the extra time required by the simulator for each event on top of actual processing time. Tests show that for simulations involving event processing times greater than 1000 μsec, the analyser provides good results. For shorter events (10-40 μsec) the analyser accuracy decreases due to effects of ignored execution details. It is yet to be determined how short the event processing time can be for accurate prediction.

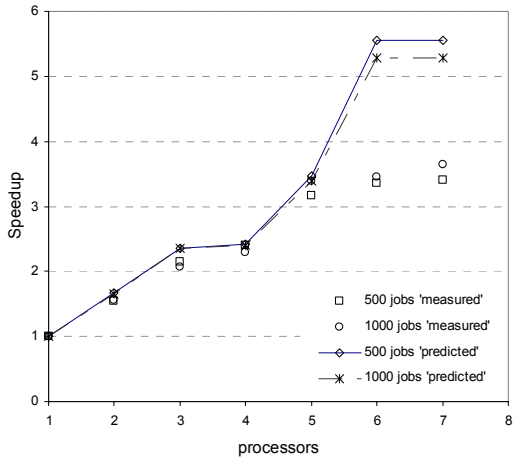

Figure 17. Execution of the sample simulation on three processors using the ideal partitioning algorithm,

Table 2. Relative prediction error for a 7-process parallel simulation.

processors	Prediction error [%]	
	500 jobs	1000 jobs
1	0	0
2	8	7
3	10	14
4	1	4
5	10	0
6	66	53
7	63	45

Two shortcomings of the current implementation are related to the Visual C++ environment. Long-running low-granularity simulations may produce traces that cannot be displayed due to Windows graphics co-ordinate system limitations. A switch is provided in the user interface to disable trace drawing in these cases, but a better solution will be provided in future versions. Because reading large traces from file can be very time consuming, it is planned to optimise this process and to overlap event input and analysis to reduce processing time.

7. CONCLUSIONS AND FUTURE WORK

This paper described a posteriori performance analyser and prediction tool that provides an easy and quick way for developers to analyse and predict the behaviour of parallel discrete-event simulation programs based upon sequential run-time trace information. We believe that such a tool can help programmers in choosing the best approach for an efficient parallel implementation of a given simulation program. Moreover, the analysis can be carried out without the actual development of the parallel simulation program.

The features of the implemented design and the resulting architecture of the analyser program were described in this paper. The program predicts the parallel execution time of a simulation based on user-defined parameters, such as the parallel simulation protocol, the number and speed of the target processors, communication parameters and topology of the interconnect. When the number of processors is smaller than the number of processes, several mapping algorithms can be used to identify the most optimal allocation of processes.

The program has an object-oriented design and implementation, making future extensions of the program with further simulation protocols or mapping algorithms a relatively straightforward process.

The program demonstrated good analysis speed and accuracy although more work is needed for the full validation of its prediction accuracy.

In the future, the system will be extended adding optimistic protocol variants and protocols for simulation programs based on Petri nets. Other areas of future work include the development of more effective process-processor allocation algorithms and a general optimisation of the software to increase its functionality, performance and accuracy.

ACKNOWLEDGEMENTS

This research has been supported by the Hungarian Ministry of Education under Grant No. FKFP-0035/2000 and by the Hungarian Scientific Research Fund (OTKA) under Grant No. F 032155.

The experiments and validations involving parallel executions of simulations were carried out on a 64-processor Sun Enterprise 10000 computer in the Hungarian Supercomputing Centre. Their support is gratefully acknowledged.

The authors are grateful for the anonymous reviewers for their invaluable help in suggesting ways to clarify the content and presentation of the paper.

REFERENCES

Bokhari S. H. 1981a, “On the Mapping Problem”, *IEEE Trans. Computers*, Vol. c-30, No. 3, March 1981.

Bokhari S. H. 1981b, “A Shortest Tree Algorithm For Optimal Assignments Across Space and Time in a Distributed Processor System”, *IEEE Trans. Software Engrg*. SE-7, 6 (Nov. 1981).

Chen S. and Eshaghian M. M. 1995, “A Fast Recursive Mapping Algorithm”, *Concurrency: Practice and Experience* Vol. 7(5), August 1995.

Clement M. J. and Quinn M. J.1993, "Analytical Performance Prediction on Multicomputers". In *Proc. Supercomputing '93*: Portland, Oregon, November 15–19, 1993, pages 886–894, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1993. IEEE Computer Society Press.

Coffman, E. G. Jr., Garey M. R. and Johnson D. S. 1978, "An Application of Bin-Packing to Multiprocessor Scheduling", *SIAM J. Comput*. 7 (1978), 1-17.

Crovella M. E. and LeBlanc T. J. 1993, *The Search for Lost Cycles: A New Approach to Parallel Program Performance Evaluation*. Technical Report TR479, University of Rochester, Computer Science Department, Dec. 1993.

Felderman R. and Kleinrock L. 1992, "Two Processor Conservative Simulation Analysis". In *6th Workshop on Parallel and Distributed Simulation (PADS92)*, pages 169–177, 1992.

Ferscha A. and Tripathi S. K. 1994, *Parallel and Distributed Simulation of Discrete Event Systems*. Technical Report CS-TR-3336, University of Maryland, College Park, Aug. 1994.

Fujimoto R. 1988, "Lookahead in Parallel Discrete Event Simulation". In *Proc. of the International Conference on Parallel Processing III*, pages 34–41, 1988.

Fujimoto R. M. 1990, "Parallel Discrete Event Simulation". *Communications of the ACM*, 33(10):30–53, October 1990.

Fujimoto R. M. 2000, *Parallel and Distributed Simulation Systems*. John Wiley and Sons, 2000.

Gelenbe E. 1989, *Multiprocessor Performance*. Series on Parallel Computing. John Wiley and Sons, New York, 1989.

Grama A. Y., Gupta A. and Kumar V. 1993, "Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures". *IEEE parallel and distributed technology: systems and applications*, 1(3):12–21, Aug. 1993.

Gupta A., Akyildiz I. F., and Fujimoto R. 1991, "Performance Analysis of Time Warp with Homogenous Processors and Exponential Task Times". In *ACM SIGMETRICS, Performance Evaluation Review*, pages 101–110, 1991.

Heiss H. U. and Dormanns M. 1996, "Partitioning and Mapping of Parallel Programs by Self-Organization", *Concurrency: Practice and Experience* Vol. 8(9), November 1996.

Jefferson D. and Sowizral H. 1985, "Fast Concurrent Simulation Using the Time Warp Mechanism". In *Proc. of the Conference on Distributed Simulation*, pages 63–69, July 1985.

Lavenberg S., Muntz R., and Samadi B. 1983, "Performance Analysis of a Rollback Method for Distributed Simulation". In *A. K. Agrawala and S. K. Tripathi, editors, Performance '83*, pages 117–132. North Holland, New York, 1983.

Lim C.-C., Low Y.-H., Gan B.-P., Jain S., Cai W., Hsu W.J. and Huang S.Y. 1999, "Performance Prediction Tools for Parallel Discrete-Event Simulation". in *Proc. PADS '99*, pp 148–155.

Lin Y. B. and Lazowska E. D. 1989, *Exploiting Lookahead in Parallel Simulation*. Technical Report 28-10-06, University of Washington, Seattle, 1989.

Lubachevsky B. D. 1988, "Bounded Lag Distributed Discrete Event Simulation". In *Proc. of the SCS Multiconference on Distributed Simulation*. The Society for Computer Simulation, 1988.

Mitra D. and Mitrani I. 1984, "Analysis and Optimum Performance of Two Message-Passing Parallel Processors Synchronized by Rollback". In *Performance '84*, pages 35–50. North-Holland, 1984.

Nandy B. and Loucks W. M. 1993, "An Algorithm For Partitioning and Mapping Conservative Parallel Simulation onto Multicomputers", In *Proc. 7th Workshop on Parallel and Distributed Simulation*, May 1993, pp 43-51.

Nicol D. M. 1993, "The Cost of Conservative Synchronization in Parallel Discrete Event Simulations". *Journal of the Association for Computing Machinery*, 40, 2:304–333, 1993.

Nicol D. M. and Mao W. 1995, "Automated Parallelization of Timed Petri-Net Simulations", *Journal of Parallel and Distributed Computing* 29, 1995, pp 60-74.

Page E.H. and Nance R.E. 1994, *Parallel Discrete Event Simulation: A Modelling Methodological Perspective*, Virginia Polytechnic Inst. and State University Technical Report, TR-94-05.

Page E.H. 1999, "Beyond Speedup: PADS, the HLA and Web-Based Simulation", In *Proc. PADS '99*, pp 2–11.

Shen H. 1992, "Self-adjusting Mapping: A Heuristic Mapping Algorithm for Mapping Parallel Programs on to Transputer Networks", *The Computer Journal*, Vol. 35., No. 1., 1992.

Srinivasan S. and Reynolds P.F. 1993, *On Critical Path Analysis of Parallel Discrete Event Simulations*, Computer Science Report No. TR-93-29.

Stone H. S. 1977, "Multiprocessor Scheduling with the Aid of Network Flow Algorithms", *IEEE Trans. Software Engrg*. SE-3, 1 (Jan. 1977).

Towsley D. F. 1987, "Allocating Programs Containing Branches and Loops Within a Processor System", *IEEE Trans. Software Engrg*. SE-12, (1987).

Wagner D. B. and Lazowska E. D. 1989, "Parallel Simulation of Queueing Networks: Limitations and Potentials". In *Proc. of 1989 ACM SIGMETRICS and PERFORMANCE '89*, volume 17,1, pages 146–155, May 1989.

Wing A. J.1992, "Discrete Event Simulation in Parallel". In *Advances in Parallel Algorithms*, Edited by Lydia Kronsjo and Dean Shumsheruddin. Halsted/Wiley, 1992.

Wong Y.-C., Hwang S.-Y., and Lin J. Y.-B. 1995, "A Parallelism Analyzer for Conservative Parallel Simulation". *IEEE Transactions on Parallel and Distributed Systems*, 6(6):628--638, June 1995.

Woodside C. M. and Monforton G. G. 1993, "Fast Allocation of Processes in Distributed and Parallel Systems", *IEEE Trans. Parallel and Distributed Systems*, Vol. 4, No. 2, February 1993.

BIOGRAPHY

ZOLTAN JUHASZ is a senior lecturer in the Department of Information Systems at the University of Veszprem (Hungary) where he leads parallel and distributed research. In the past he also held positions in The Queen's University of Belfast and the University of Exeter (UK). He received his MEng in Electrical Engineering and PhD in Computer Science from the Technical University of Budapest (Hungary). His current research interests are in the performance prediction of parallel – particularly discrete event simulation – programs, and in the design and performance evaluation of large grid and multi-agent systems.

STEPHEN TURNER joined Nanyang Technological University (Singapore) in 1999 and is currently an Associate Professor in the School of Computer Engineering and Director of the Parallel and Distributed Processing Laboratory. Previously, he was a Senior Lecturer in Computer Science at Exeter University (UK). He received his MA in Mathematics and Computer Science from Cambridge University (UK) and his MSc and PhD in Computer Science from Manchester University (UK). His current research interests include: parallel and distributed simulation, distributed virtual environments, grid computing and multi-agent systems.

KRISZTIAN KUNTNER is a PhD student in the Department of Information Systems at the University of Veszprem (Hungary), where he received his MSc in Information Tecnnology in 2002. His main research interests are in large-scale distributed systems, wide-area service discovery, and performance prediction of discrete event simulations.

MIKLOS GERZSON is a senior lecturer in the Department of Automation at the University of Veszprem (Hungary). He received his MSc degree in Chemical Engineering from the University of Veszprem, and his PhD in Chemistry from the Hungarian Academy of Science. His research interests include modelling and control of different systems with the emphasis on process systems and parallel computing. His teaching activity is related to these fields and measurement techniques at the University of Veszprem and University of Pecs (Hungary).