Performance Analysis of Distributed Object-Oriented Applications

Final Report

NASA Grant NCC3-572

Period April 1, 1998 to December 31, 1998

James D. Schoeffler
Professor of Computer Science
Cleveland State University
Cleveland, Ohio 44115

Performance Analysis of Distributed Object-Oriented Applications

Summary of Research

The purpose of this research was to evaluate the efficiency of a distributed simulation architecture which creates individual modules which are made self-scheduling through the use of a message-based communication system used for requesting input data from another module which is the source of that data. To make the architecture as general as possible, the message-based communication architecture was implemented using standard remote object architectures (CORBA and/or DCOM). A series of experiments were run in which different systems are distributed in a variety of ways across multiple computers and the performance evaluated. The experiments were duplicated in each case so that the overhead due to message communication and data transmission can be separated from the time required to actually perform the computational update of a module each iteration. The software used to distribute the modules across multiple computers was developed in the first year of the current grant and was modified considerably to add a message-based communication scheme supported by the DCOM distributed object architecture.

The resulting performance was analyzed using a model created during the first year of this grant which predicts the overhead due to CORBA and DCOM remote procedure calls and includes the effects of data passed to and from the remote objects. A report covering the distributed simulation software and the results of the performance experiments has been submitted separately and is titled: "A Component-based Distributed Simulation Architecture and its Performance", J. D. Schoeffler, December 28, 1998.

The above report also discusses possible future work to apply the methodology to dynamically distribute the simulation modules so as to minimize overall computation time.

NASA GRANTEE
NEW TECHNOLOGY REPORT

NASA requires each research grantee, research contractor, and research subcontractor to report new technology to the NASA Technology Utilization Office. The required reports and corresponding schedules are as follows:

Title of Report	Form Number	Timetable
Individual Disclosure	NASA 666A	The grantee discloses each discovery of new technology individually, at the time of its discovery.
Interim Report	NASA C-3043	For multi-year grants, the grantee summarizes the previous year's disclosures on an annual basis. The first Interim New Technology (NT) Report is due exactly 12 months from the effective date of the grant.
Final Report	NASA C-3043	The grantee submits a cumulative summary of all disclosed discoveries. This Final NT Report is submitted immediately following the grant's technical period of performance.

Grantee Name and Address: James D. Scheffler
Cleveland State University
Cleveland Ohio 44115

Report Submitted by: J.D. Scheffler

Telephone Number: (216) 687-4755

NASA Grant Title: Performance Analysis of Distributed Object-Oriented Applications

NASA Grant Number: NCC 3-572

NASA Project Manager: Gregory Follen

Grant Completion Date: 12/31/98

Today's Date: 12/28/98

New technology may be either reportable items or subject inventions.

A reportable item is any invention or discovery, whether or not patentable, that was conceived or first actually reduced to practice during the performance of the grant, contract or subcontract. Large business contractors and subcontractors must disclose reportable items as they are discovered and submit a noncumulative list of these new technology items on an annual basis [ref: Interim NT Report] and a cumulative list at the completion of the grant, contract (or subcontract) period [ref: Final NT Report].

A subject invention is any invention or discovery, which is or may be patentable, that was conceived or first actually reduced to practice during the performance of the contract or subcontract. Grantees, small business contractors and subcontractors must, at a minimum, disclose subject inventions as they are discovered and submit a cumulative list of these new technology items on an annual basis [ref: Interim NT Report] and at the completion of the grant, contract (or subcontract) period [ref: Final NT Report].

Grantees, small business contractors and small business subcontractors are only required to disclose and report patentable items (subject inventions). However, we encourage that grantees, small business contractors and small business subcontractors disclose both patentable and nonpatentable (reportable) items, both of which are automatically evaluated for publication as NASA tech briefs and considered for NASA *Tech Brief* awards.

PLEASE COMPLETE THE REVERSE SIDE OF THIS FORM AND MAIL TO THE FOLLOWING ADDRESS:

NASA Lewis Research Center
Attn: Kathy Kerrigan
Technology Utilization Office; Mail Stop 7-3
Cleveland, Ohio 44135

I. General Information

1. Type of Report: ( ) Interim (√) Final

2. Size of Business: ( ) Small ( ) Large (√) Nonprofit Organization

3. Have any nonpatentable items resulted from work performed under this subcontract during this reporting period?
   ( ) yes (√) no

4. Have any subject inventions resulted from work performed under this subcontract during this reporting period?
   ( ) yes (√) no

5. Are new technology items (nonpatentable or patentable) being disclosed with this report?
   ( ) yes (√) no

II. New Technology Items

Please provide the title(s) of all new and previously disclosed new technology items conceived or first actually reduced to practice under this grant.

Title	Internal Docket Number	Patent Appl. Filed	Patentable Item	Nonpatentable Item
1. _____	_____	( )	( )	( )
2. _____	_____	( )	( )	( )
3. _____	_____	( )	( )	( )
4. _____	_____	( )	( )	( )

III. Subcontractors

Please complete the following section listing all research subcontractors participating to date. Include each subcontractor's name, address, contact person, and telephone number.

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

_____ _____

IV. Certification

I certify that active and effective procedures ensuring prompt identification and timely disclosures of reportable new technology items have been followed. Furthermore, I certify that all new technology items required to be disclosed and conceived during the period identified on this form, have been disclosed to NASA.

_____

Name and Title of Authorized Official

_____ _____

Signature and Date **Grant Monitor**

GOVERNMENT-FURNISHED EQUIPMENT INVENTORY

Grant No.: NCC 3-572

ORS No:

Principal Investigator: James D Schaeffer

Grant Expiration: 12/31/98

Status: **Final**

DISPOSITION INSTRUCTIONS

☐ **RETURN**

☐ **TRANSFER TO ANOTHER GRANT**

☐ **RETAIN**

Description	Model#	Furnished Date	Equip Cost	*Cond. Code	Location
None					

James D Schaeffer 12/28/98

(I, the principal investigator, certify that the above information is correct) (Date)

*Equipment Condition Codes:

4= Good
5 = Fair
6=Poor

7=Repairs Req'd (<15% of Acq Cost)
8=Reparis Req'd (16-40% of Acq Cost)
9=Repairs Req'd (41-65% of Acq Cost)

X=Salvage (>65% of Acq Cost)
S=Scrap(Value is material Content only)

A Component-based Distributed Simulation Architecture and its Performance

James D. Schoeffler
Department of Computer Science
Cleveland State University

NASA Grant NCC 3-572

Summary

The purpose of this report is to evaluate the efficiency of a distributed simulation architecture which creates individual modules which are made self-scheduling through the use of a message-based communication system used for requesting input data from another module which is the source of that data. To make the architecture as general as possible, the message-based communication architecture is implemented using standard remote object architectures (CORBA and/or DCOM). This report describes the simulation architecture and presents a series of experiments in which different systems are distributed in a variety of ways across multiple computers and the performance evaluated. The experiments are duplicated in each case so that the overhead due to message communication and data transmission can be separated from the time required to actually perform the computational update of a module each iteration.

1 Overview of the distributed simulation architecture

The objective of a distributed computation for simulation of an aircraft engine is to perform computation in parallel on a set of processors coupled across a local area network. The engine may be characterized by a set of inter-related components with the simulation being carried out either for a series of time steps or a series of iterations or both. In order to gain flexibility in changing component designs, it is desirable that components be assignable to any processor in the set with no master control scheduling the execution and data exchange of each component. An advantage of this is the ability to base scheduling dynamically on the relative processor needs of each component.

The distributed simulation investigated in this report achieves this scheduling independence by representing components as modules with inputs and outputs each of which is a set of variables. An output of one component is coupled via connectors to an input of another component. This implies that the first component must calculate its output variables each iteration or time-step before the second component can use those values as input to is computation in the corresponding iteration or time-step.

A cooperative synchronization of the component executions is achieved by required that components request their inputs and wait until they have been received before executing. In this way, modules execute as soon as they are ready and upon completion of execution, send outputs which in turn trigger executions of other components waiting for their inputs.

An expected payoff of this architecture is the ability to adapt or change a distribution during the running of a simulation and to base the choice of distribution of components upon current and past execution time requirements.

1.1 Solvers and modules and their design constraints

Components are represented by objects whose class (in the C++ sense) provides only the behavior of the component (code) related to the initialization of a component and the carrying out of the execution associated with an iteration or time-step. All other behavior is inherited from a base class which handles all acquiring of input data, triggering of the update execution, and sending of outputs to other modules.

The intent here is to permit the creation of component code related to the specific function of the component and not to have to consider the details of how the simulation actually takes place.

Two kinds of component base classes are provided: the "module" class which is associated with an actual engine component and the "solver" class which is associated with the requirements of a conventional solver, namely, the sending of inputs to one or more components to start an iteration (time-step), the receiving of the components' outputs, and the recalculation of inputs to be sent to start the next iteration (time-step).

An interconnection of a set of modules which causes a closed loop must be broken by the insertion of a solver component to prevent a deadlock that would occur as every module in the loop requests its input from the previous component. This leaves all of the modules waiting for input and unable to compute. Solvers can also be used to control the rate at which convergence in an iteration takes place.

1.2 Components and inter-component communication

Each component input is attached to a connector which is connected to another connector that in turn is connected to an output of another component. Hence each component is directly connected to one connector for each of its inputs and its outputs and these connectors are in the same process as the component itself. Connectors are created as the simulation is initialized and distributed. Each connector is provided with the location of its mate connector (the connector it is connected to) and whether the data it is connected to a module input or a module output.

Thus the requesting of an input (for a specific iteration number or time-step number) involves the associated connector requesting that data through its mate connector which in turn records the request with the component supplying that data. The connectors hide the details of whether the two connectors are in the same process, separate processes on the same processor, or on separate workstations on the network. Upon creation, they

discover the type of connection and use the most appropriate implementation for the requesting of data and the sending of output data.

1.3 Message-based inter-process communcation

Since components may request inputs from multiple source components, it is desirable to make the handling of requests and replies as concurrent as possible so as not to place bottlenecks in the path of the computation. To this end, each component is provided a message queue in which messages (requests for data or replies containing data) are placed. In order to make the message system as machine independent and operating-system independent as possible, it is itself implemented using distributed-object technology, CORBA or DCOM. Thus a messaging interface is defined and that interface on each message queue made available to all the components. Because of the characteristics of CORBA and DCOM, multiple components in separate machine can safely use their interface to pass a message to a single component without any concurrency problem. Using these essentially standard disributed technologies, it is expected that the systems will be less dependent upon computer vendor, operating system version, etc.

1.4 Module group architecture

To minimize any overhead, components may be grouped into a "module group" which is executed by a single process. A module group may contain one or many modules or solvers in it, and has one message queue which serves all the modules and servers within it. The module group is inside a CORBA/DCOM object whose interface reference or pointer is provided to all other module groups.

Within a module group, components can communicate with very small overhead: messages go directly from component to component without passing through any message queue.
Since the module group is a process, only one module can ever execute at any instant. Hence a group which contains two modules, for example, is best allocated to modules which can't execute concurrently (e.g., the input of one module comes from an output of the other module).

Module groups are created dynamically from a specification of module interconnections and a choice of distribution. In the current implementation discussed in this report, a module group cannot be modified during a run. Hence modules cannot move from module group to module group during a given simulation run. They can move between runs. The design is such that the implementation could be easily modified to allow them to move during a run but this is left to a future effort.

Modules in separate module groups transparently use the CORBA/DCOM technology for sending and receiving messages and hence incur the overhead of such a function call. The overhead in these calls has been measured and modeled [Schoeffler, 1998] and is presented later in this report.

1.5 Module group scheduler

Within each module group (process) a group scheduler allocates the process in a run-to-complete mode to the following priorities.

First, any data messages in the message are delivered to the waiting module or solver component. This involves copying the message data into a pre-allocated data object for each input of each component. In this way, the module does not have to be active to receive a data message which has caused it to block until it is received.

Second, any request message is passed to a receive function of the associated component. If the request is for data that is not yet available, the request is simply recorded for later delivery by the component. If the data is available, the object immediately sends the data via a message through its connector. In either case, the state of the component is updated. For example, the arrival of the last input data message changes the component into a ready-to-execute state. Similarly, the arrival of the last request for output data to a component which has executed and is waiting to deliver its outputs causes it to change to the next iteration (time-step) and be ready to run (to request inputs for the next iteration).

Third, a ready-to-run module is allowed to execute. It may request inputs or perform its update calculation for the current iteration or time-step. ready components are scheduled run-to-completion on a first-come-first-served basis.

1.6 Architecture of groups within a processor

Each processor may contain multiple module groups, each of which is created as a separate DCOM or CORBA object. Each module group is allocated its own thread of execution that is scheduled by the machine operating system on a time-shared basis.

All messages arrive to the process main thread which then deposits the messages in the message queue of the appropriate module group. The message queues represent the only place in the overall design where critical-section processing is required in order to prevent concurrency problems (e.g., one thread adding a message to a queue and another thread removing a message from a queue at the same time). This also means that the only information sharing among threads is message data. In particular, threads do not attempt to pass through different module groups so that code has to contain critical sections with their attendant overhead.

1.7 Network architecture

The processors are connected into a local area network. The details of the network are transparent to DCOM and CORBA. The current implementation uses one application program to start up a simulation by reading connection and distribution information from a file and then creating and interconnecting the module groups in the various machines. It is expected that the interconnection information would normally come from an interactive

user via a GUI (graphic-user-interface) and the distribution information would optionally come from the interactive user or automatically from a distribution-scheduler which maintained past history of compute times of the various modules and adapted as the simulation proceeds.

2 Performance of the architecture

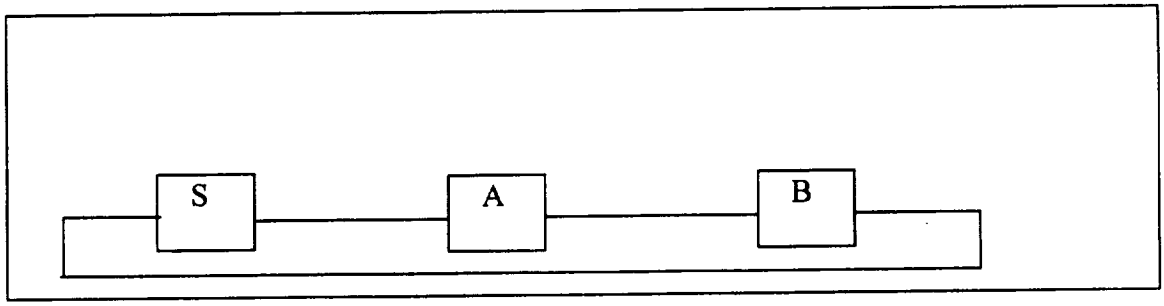
2.1 Potential parallel execution of solvers and modules

The architecture treats each module as though it were executing independently of the others by structuring it so that it passes sequentially through a series of states which result in its coordination with other objects no matter where they execute. These states are:

State 1. Starting an iteration
Request outputs from other modules which are inputs to this module. The requests are for the specific iteration desired.
Proceed to state 2 only after all requested inputs have actually been received.
State 2. Updating the outputs of the module
Perform the computation associated with this iteration using the inputs received.
Upon completion, go to state 3.
State 3. Sending results to modules which take this module's output as one of their inputs
Send output data to all request already received.
The number of modules that use outputs of this modules as their inputs is known. Wait for any additional request for data are received and then send output data to that module.
When the output has been sent to the last module, proceed to the next iteration in state 1.

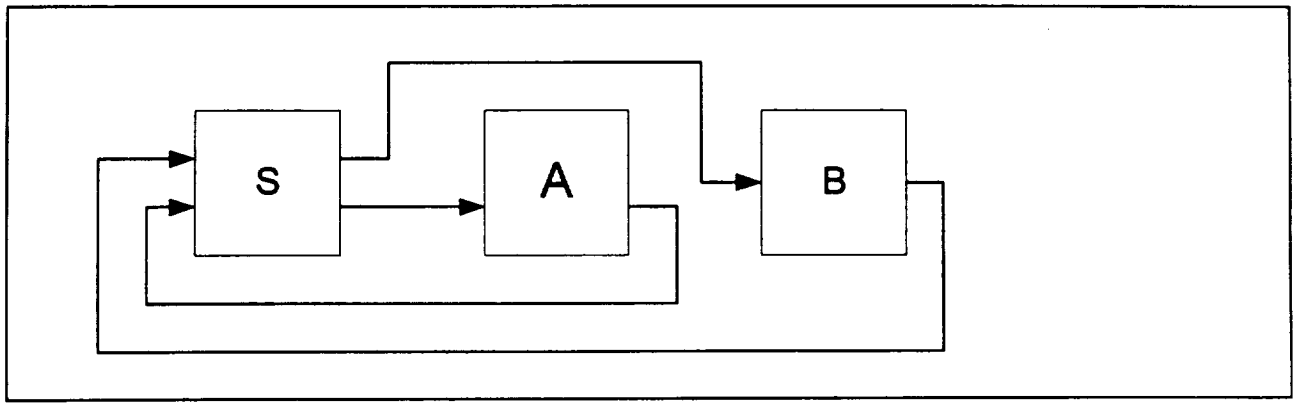
Clearly no module can begin its computation until all its required inputs are present but may begin as soon as the last input arrives. If there is only one module assigned to execute in a given processor, then it will begin computation as soon as its inputs have arrived. If there are multiple modules in a processor, then the modules that are ready to compute must compete for the processor. The performance of a distributed set of modules is then very much dependent upon how they are allocated to processors for execution.

Consider the following simple example of one solver (S) and two modules (A and B):



This system is such that no parallelism is possible. At the start of iteration k, module A requests its single input from the Solver (its output for iteration k) which immediately provides it. Concurrently, B requests its input from module A (its output for iteration k) which cannot provide it yet. Once the solver S has sent its output, it immediately requests its input from module B (its output for iteration k) which cannot provide it. Only module A can execute. Upon completing its computation, it immediately provides its output to module B (because it has previously received a request). Module B can then execute in iteration k and immediately provides its output to the solver (again because it has previously received the request). Now only the solver S can execute. Thus there is no incentive to allocate these modules to separate machines because of the way the solver and modules have been structured.

Consider an alternate version of this example as shown below:



Now the solver has two inputs and two outputs. Hence the requests by modules A and B can both be satisfied at the beginning of the iteration. Hence both A and B can begin their execution for this iteration as soon as their data is received. Of course, this will not be completely simultaneous because the solver replies to one request and then the next. While A and B are computing, S will request inputs from A and B and then wait until each completes its computation and sends its input to S. Modules A and B will also request inputs for their next iteration. Solver S will then compute and when complete, begin the next iteration by sending the computed outputs to A and B.

Now computation of A and B can be overlapped in time and it is advantageous to assign them to different processors. Note that solver S may be assigned to either processor with little effect on the concurrent processing since modules A and B always are waiting while S is computing and vice versa (except for activities such as requesting and sending messages).

Although the allocation of modules to module groups and module groups to processors is obvious in simple situations like the above, it is far less obvious in more realistic situations. The allocation is very dependent upon the length of computation time associated with each module each iteration and the amount of time required to transmit

requests for data and receive replies to those request. For the purpose of allocation, a model of the computation is desired which can be used as a basis for the allocation.

This report emphasizes the portion of the model associated with time to transmit messages using DCOM or CORBA as a basis for the transmission among processors. Since the simulation of aircraft engine models is highly iterative, it is expected that accurate computation times per iteration will be readily available and when combined with the message transmission portion of the model, can be used as a basis for allocation of modules to processors and even re-allocation to processors as the computation proceeds.

2.2 The experimental implementation

The implementation of distributed simulation architecture reported previously [schoeffler, 1997] used a message-based communication architecture developed at NASA LERC called APPL. This program was redesigned leaving the design of the module, connector, module group, and scheduling identical but replacing the APPL protocol with the message architecture implemented using object technology. Specifically, the implementation was carried out using DCOM and running on ALR multi-processor workstations.

To concentrate on the overhead introduced by the distributed architecture, the program was specialized to constrain all modules and solver components to have the same update time (execution time to calculate output variables once input variables were available). In the experimental runs, this common update time could be set to zero or some arbitrary time. When the update time is set to zero, the elapsed execution time includes the time to prepare, send, and receive messages. Furthermore, the inputs and outputs were constrainted to all be arrays of doubles with identical array sizes. This size could be set at at any size in the range zero to 1000 and was varied. For a zero length data array, there is no message preparation overhead (e.g., packing data values into a message) and hence all elapsed time is pure overhead time. Note that by assigning all modules to the same module group, there is no message-related overhead at all nor is there any overhead at all due to the use of DCOM or CORBA. Then by varying message length for the same compute time and module distribution, the effects of message size can be easily determined.

2.3 Message overhead (DCOM)

A model of CORBA and DCOM overhead involved in passing messages which takes into account the machines involved and the actual argument types and sizes was created previously [Schoeffler,1998]. This model typically provided accuracy on the order of 15%-20%. It attempts to take into account both operating system overhead, message marshalling overhead (dependent upon argument sizes and types), and direction of message transfer. The latter includes [out] -- sending data from the source to the function called via DCOM or CORBA; [in]-- sending no data to the function but receiving data

back from the function; and [in-out] -- sending data to the function and also receiving data back again.

In the above report, the overhead model is taken to be:

$$O(m_1,m_2,k,r) = T(m_1,k,r) + T(m_2,k,r) + T_n$$

where $T(m_j,k,r)$ denote the portions of a function-call overhead within machine *j* and T_n is the network transmission time (for a specific network *n*) and is non-zero only for remote calls. The linear approximation to the observed overhead gives

$$T(m,k,r) = C(m,k,r) + L*V(m,k)$$

where:

C(m,k,r) is a constant dependent upon the machine m, the argument type k, and the call-type r

L is the length in bytes of the argument passed in both directions. This means it is the argument length for an input argument or an output argument but is twice the argument length for an input-output argument.

V(m,k) is a constant dependent upon the machine m and argument type k but not the call-type (local or remote). V has units of milliseconds per byte since L (argument length) is in bytes. It is convenient to use units of milliseconds-per-kilobyte because of the size of V. This of course would change the term to 0.001*L*V(m,k);

The network transmission overhead is

$$T_n = L*S_n$$

where L is the argument length as before and S_n is the transmission time of the network in milliseconds per byte (e.g., 0.0008 milliseconds/byte or 0.8 milliseconds/kilobyte for a 10MHz network). The actual message passed across the network must include some additional bytes (e.g., for the message header) but this time is dependent upon the details of the network protocol and the model assumes it is included in the fixed-time portion of the model.

Notice that both the fixed and length-dependent overhead terms are incurred at both the client (function-call source) and server (destination executing the function itself).

There are 3 parameters per argument type or a total of 9 parameters but each computer may have different values for these parameters. Hence the parameters must be determined from experiments involving the different kinds of computers of interest.

The underlying component technology (DCOM or CORBA) is an additional factor of course so models are presented for both technologies for machines supporting both DCOM and CORBA (Intel machines running the NT4 operating system) but only for CORBA for other machines (UNIX machines).

The parameters reported in [xxxxxxxxxxx] for the COM/DCOM technology in Intel 200 MHz machines are the following:

	Fixed Local (MS)	*Fixed Remote (MS)*	*Variable (MS/KB)*
input argument	0.10	0.39	0.75
output argument	0.11	0.42	1.08
input-output argument	0.10	0.38	0.79

The model evaluated in this paper uses DCOM calls to request data and to send the requested data. Because the source of the data may or may not be ready to send the data at the moment it is requested, two separate calls are made: one to request the data, and one to send the data. The request call contains no data and hence is equivalent to a single zero-length argument. The reply call does contain data whose size can vary considerably. In all cases however, the call consists of a single argument with an input-type argument. Furthermore, all experiments were done using a group of identical Intel processors connected on a 10 MHz either net. Hence the overhead models of interest reduce to the following:

O(Local Request Messsage) = 2*0.10 ms
O(Local Reply Message, L bytes) = 2*0.10 + 2*0.75*L/1000 ms

O(Remote Request Messsage) = 2*0.39 ms
O(Remote Reply Message, L bytes) = 2*0.39 + 2*0.75*L/1000 + 0.0008*L ms

Here the parameters are from the input-argument line of the above table. The length-dependent parameter has units of milliseconds per kilobyte. Since the length L is in bytes, a factor of 0.001 appears in the equation.

These equations further simplify to the following:

O(Local Request Messsage) = 0.20 ms
O(Local Reply Message, L bytes) = 0.20 + 0.0015*L ms

O(Remote Request Messsage) = 0.78 ms
O(Remote Reply Message, L bytes) = 0.78 + 0.0095*L ms

Experiments were run with message data of 4 different lengths: 4, 100, and 1000 doubles or lengths of 32, 800, and 8000 bytes. Using the above model equations, the overhead times were computed and are shown in the following table:

Data Length (bytes)	Local (ms)	Remote (ms)
0	0.20	0.78
32	0.25	1.08
800	1.40	8.38
8000	12.20	76.78

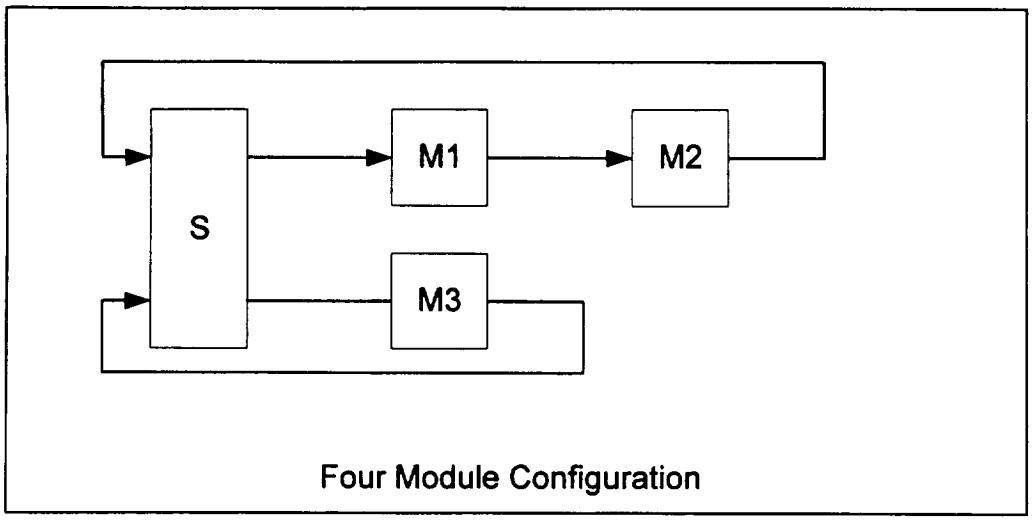
In this table, the first row corresponds to messages with no data (request messages). A module with two local inputs requires two request messages which take 0.20 ms each. The local modules supplying those inputs each take 1.40 ms for an 800 byte messsage (an array of 100 doubles) or 12.20 ms each for an array of 1000 doubles. If the same module was getting its inputs from two remote modules, the request messages are about 4 times longer and the data messages about 6 times longer.

The total of two request messages and two reply messages would be 24.8 ms or 155 ms for the 1000-double array for the local and remote cases. If the module computation time per iteration were 1 second per iteration, these overhead times are 3% and 16% respectively. For long computation times, the message overhead is not important and only the actual amount of overlapped computation is important to the response time for a set of modules. For module computation times of 0.25 seconds, however, the message overhead percentage jumps to 12% and 64% respectively. Clearly, remote processors lead to overtime comparable to the overlap time. This is especially true when the various modules have a wide distribution of computation time per module.

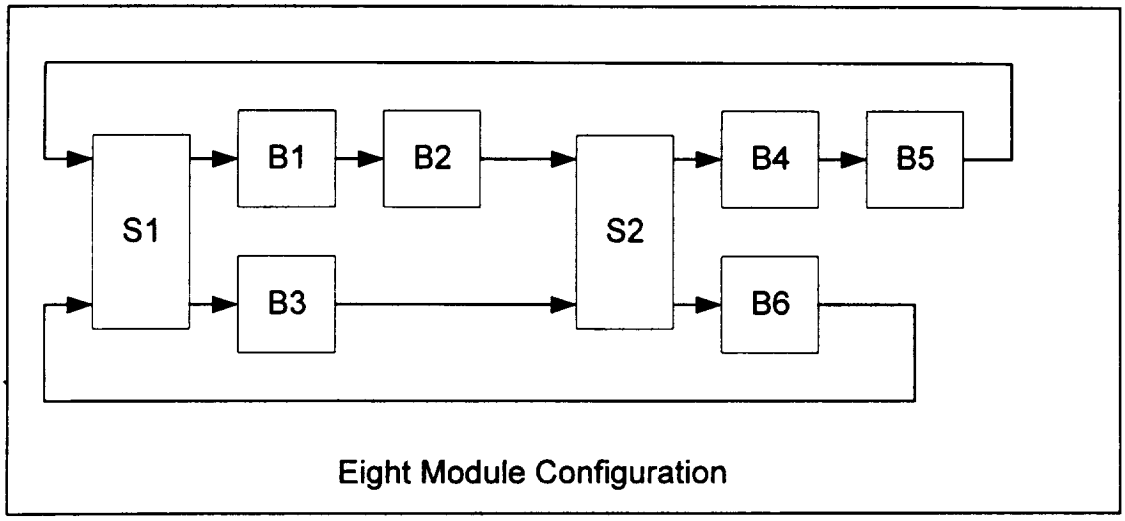
2.4 Experimental configurations

Two basic simulation configurations were experimentally investigated, each for a variety of different data object sizes, two different update times (0 and 0.719 seconds), and for a variety of different allocations among two network-coupled workstations each containing four processors. This does not cover the more practical case where the updates times of the modules per iteration vary significantly but is useful to see the effects of parallelism and message overhead.

The two configurations are shown below:



Four Module Configuration



Eight Module Configuration

Notice that solvers have been used in both configurations to break closed loops of modules, one in the four-module configuration and two in the eight-module configuration.

Possible parallelism in the four module configuration is limited two modules B1 and B3 executing in parallel (because both get their input from the solver). Module B2 cannot execute until B1 completes. The solver cannot execute until both B2 and B3 complete. Hence for equal 1 second update times and no overhead, this configuration would require 3 seconds to execute each iteration provided modules B1 and B3 are allocated to two different processors (so they can execute concurrently). If they are allocated to the same processor, then there is no parallelism and the configuration would require 4 seconds to

execute one iteration assuming a one second update time for each module. Thus this configuration allows a maximum reduction of 25% (4 seconds down to 3 seconds).

Maximum parallelism in the eight module configuration is easily seen to be 3 seconds provided modules B1, B3, B4, and B6 execute concurrently on 4 processors, modules b2 and B5 execute concurrently on 2 processors, and the two solvers execute concurrently on two processors. This configuration allows a maximum reduction of 8 seconds down to 3 seconds or 63% assuming no overhead.

2.5 Experimental results

2.5.1 Example 1: The Four Module Configuration

The Four Module Configuration contains one server (two inputs and two outputs) and three modules. The configuration and results of the experiments are shown in the tables on the following pages for the case of 4 doubles of data per output, 100 doubles, and 1000 doubles. Each case is shown on two pages. Each of the seven data columns corresponds to a different experiment involving the distribution of the four modules.

The first 10 rows contain information about the modules and the parameters of the DCOM performance model used in succeeding rows. The rows "Modules/Process" and "Processes/Machine" indicate the distribution of the modules among processes and machines. For example, the first data column shows that there is 1 module per process and 2 processes per machine. Hence this experiment involves 2 machines each containing 2 processes. There can be no parallelism of the two processes in a machine but it is possible for there to be parallel computation in the two machines. The fourth data column shows all modules in 1 processes and hence this represents an experiment with one machine and one process. This configuration allows no parallel computation. The succeeding lines display the results of each experiment: the total execution time with the computational update time of a module being zero seconds or 0.719 seconds; the number of iterations; and the time per iteration.

Of special interest is the pair of lines, Time/Iteration 0 sec and 0.719 seconds. The former is all overhead and data transmission time. The latter includes the large update time per module (and solver).

The remainder of the data shows for each process which modules it contains, the machine in which it executes, and the total message load per iteration. The table continues on the next page with the top portion duplicated for readability. The remaining rows reduce the data. Calculated are the iteration times for all update computation done serially and for all done in parallel. The actual time is show for comparison. Then the DCOM performance model is used to calculate the update times for the various message types (the overhead

and data transmission times). The second last line shows the addition of the estimated serial overhead time to the serial module update time. The last shown shows the amount of reduction via parallel (overlapped) computation from the estimated serial computational time to the observed time. With the little opportunity for overlap with the four-module configuration, the results agree with the previously estimated 25% reduction factor.

The Four Module Configuration Results (4 bytes of data/output)

Date: 8/10/98
System:
A2->B1->B2->A2
A2->B3->A2

perf4mt -- 4 modules, multi-threaded to use ALR processors

Type	Name	#inputs	#outputs	#doubles
Server	A2	2	2	4
Module	B1	1	1	4
Module	B2	1	1	4
Module	B3	1	1	4

ALR model: T = A + B*Length							
Coefficient A (millisecs)	0.39						
Coefficient B (millisecs/kilobyte)	0.75						
Total Number of modules and servers	4						
Maximum parallel number of modules	3						
Input: Modules/Process (0=all)	1	2	3	0	1	2	3
Input: Processes/Machine (0=all)	2	1	1	0	0	0	0
# doubles in data	4	4	4	4	4	4	4
data msg size (bytes)	40	40	40	40	40	40	40
request msg size (bytes)	4	4	4	4	4	4	4
module compute time (secs)	0.719	0.719	0.719	0.719	0.719	0.719	0.719
# iterations	11	10	10	10	11	11	10
Total time (secs) - 0.719 sec comp	24.03	23.31	21.63	28.73	23.73	24.42	21.63
Total time (secs) - 0 sec comp	0.516	0.453	0.375	0.110	0.156	0.125	0.375
Total time (secs) - 0 sec comp, 100 its				0.719			
Time/iteration (secs) - 0 sec comp	0.0469	0.0453	0.0375	0.0110	0.0142	0.0114	0.0375
Time/iteration (secs) - 0.719 sec comp	2.185	2.331	2.163	2.873	2.157	2.220	2.163
# ALR1 processes	2	1	1	1	4	2	2
# ALR2 processes	2	1	1	0	0	0	0

Process 1							
Modules in process	A2	A2,B2	A2,B2,B3	A2,B2,B2,B3	A2	A2,B2	A2,B2,B3
Machine	1	1	1	1	1	1	1
#total requests	11	20	10	0	0	0	0
#total data sends	22	20	10	0	0	0	0
#requests/iteration	1	2	1	0	0	0	0
#data sends/iteration	2	2	1	0	0	0	0
Process 2							
Modules in process	B2	B1,B3	B1	B2	B1,B3	B1	
Machine	1	2	2		1	1	1
#total requests	11	20	10		0	0	0
#total data sends	0	20	10		0	0	0
#requests/iteration	1	2	1	0	0	0	0
#data sends/iteration	0	2	1	0	0	0	0
Process 3							
Modules in process	B1			B1			
Machine	2				1		
#total requests	11				0		
#total data sends	11				0		
#requests/iteration	1	0	0	0	0	0	0
#data sends/iteration	1	0	0	0	0	0	0
Process 4							
Modules in process	B3			B3			
Machine	2				1		
#total requests	11				0		
#total data sends	11				0		
#requests/iteration	1	0	0	0	0	0	0
#data sends/iteration	1	0	0	0	0	0	0

The Four Module Configuration Results (4 bytes of data/output)

Input: Modules/Process (0=all)	1	2	3	0	1	2	3
Input: Processes/Machine (0=all)	2	1	1	0	0	0	0
# doubles in data	4	4	4	4	4	4	4
data msg size (bytes)	40	40	40	40	40	40	40
request msg size (bytes)	4	4	4	4	4	4	4
module compute time (secs)	0.719	0.719	0.719	0.719	0.719	0.719	0.719
# iterations	11	10	10	10	11	11	10
Total time (secs) - 0.719 sec comp	24.03	23.31	21.63	28.73	23.73	24.42	21.63
Total time (secs) - 0 sec comp	0.516	0.453	0.375	0.110	0.156	0.125	0.375
Total time (secs) - 0 sec comp, 100 its				0.719			
Time/iteration (secs) - 0 sec comp	0.0469	0.0453	0.0375	0.0110	0.0142	0.0114	0.0375
Time/iteration (secs) - 0.719 sec comp	2.185	2.331	2.163	2.873	2.157	2.220	2.163
# ALR1 processes	2	1	1	1	4	2	2
# ALR2 processes	2	1	1	0	0	0	0
One Iteration Compute serial (secs)	2.876	2.876	2.876	2.876	2.876	2.876	2.876
One Iteration Compute max parallel (secs)	2.157	2.157	2.157	2.157	2.157	2.157	2.157
Actual One Iteration Compute (secs)	2.185	2.331	2.163	2.873	2.157	2.220	2.163
Total No. Rem. Rqst Msgs /iteration	44	40	20	0	0	0	0
Total No. Rem. Data Msgs /iteration	44	40	20	0	0	0	0
Est. Time/Request Msg (millisecs)	0.393	0.393	0.393	0.393	0.393	0.393	0.393
Est. Time/Data Msg (millisecs)	0.42	0.42	0.42	0.42	0.42	0.42	0.42
Est Serial Msg Total Time (seconds)	0.036	0.033	0.016	0.000	0.000	0.000	0.000
Est Serial Msg Total Time/iter (seconds)	0.003	0.003	0.002	0.000	0.000	0.000	0.000
Total Serial Time/iter (seconds)	2.879	2.879	2.878	2.876	2.876	2.876	2.876
Reduction Factor	0.74	0.79	0.74	1.00	0.75	0.77	0.74

The Four Module Configuration Results (100 bytes of data/output)

Date: 8/10/98	perf4mt -- 4 modules, multi-threaded to use ALR processors						
System:	Type	Name	#inputs	#outputs	doubles		
A2->B1->B2->A2	Server	A2	2	2	100		
A2->B3->A2	Module	B1	1	1	100		
	Module	B2	1	1	100		
ALR model: T = A + B*Length	Module	B3	1	1	100		
Coefficient A (millisecs)	0.39						
Coefficient B (millisecs/kilobyte)	0.75						
Total Number of modules and servers	4						
Maximum parallel number of modules	3						
Input: Modules/Process (0=all)	1	2	3	0	1	2	3
Input: Processes/Machine (0=all)	2	1	1	0	0	0	0
# doubles in data	100	100	100	100	100	100	100
data msg size (bytes)	808	808	808	808	808	808	808
request msg size (bytes)	4	4	4	4	4	4	4
module compute time (secs)	0.719	0.719	0.719	0.719	0.719	0.719	0.719
# iterations	11	10	10	10	11	11	10
Total time (secs) - 0.719 sec comp	24.05	25.64	21.66	30.34	23.73	24.45	21.59
Total time (secs) - 0 sec comp	0.547	0.516	0.391	0.125	0.172	0.125	0.406
Total time (secs) - 0 sec comp, 100 its				1.125			
Time/iteration (secs) - 0 sec comp	0.0497	0.0516	0.0391	0.0125	0.0156	0.0114	0.0406
Time/iteration (secs) - 0.719 sec comp	2.186	2.564	2.166	3.034	2.157	2.223	2.159
# ALR1 processes	2	1	1	1	4	2	2
# ALR2 processes	2	1	1	0	0	0	0
Process 1							
Modules in process	A2	A2,B2	A2,B2,B3	A2,B2,B2,B3	A2	A2,B2	A2,B2,B3
Machine	1	1	1	1	1	1	1
#total requests	11	20	10	0	0	0	0
#total data sends	22	20	10	0	0	0	0
#requests/iteration	1	2	1	0	0	0	0
#data sends/iteration	2	2	1	0	0	0	0
Process 2							
Modules in process	B2	B1,B3	B1	B2	B1,B3	B1	
Machine	1	2	2		1	1	1
#total requests	11	20	10		0	0	0
#total data sends	0	20	10		0	0	0
#requests/iteration	1	2	1	0	0	0	0
#data sends/iteration	0	2	1	0	0	0	0
Process 3							
Modules in process	B1			B1			
Machine	2				1		
#total requests	11				0		
#total data sends	11				0		
#requests/iteration	1	0	0	0	0	0	0
#data sends/iteration	1	0	0	0	0	0	0
Process 4							
Modules in process	B3			B3			
Machine	2				1		
#total requests	11				0		
#total data sends	11				0		
#requests/iteration	1	0	0	0	0	0	0
#data sends/iteration	1	0	0	0	0	0	0

The Four Module Configuration Results (100 bytes of data/output)

Date: 8/10/98	perf4mt – 4 modules, multi-threaded to use ALR processors						
System:	Type	Name	#inputs	#outputs	doubles		
A2->B1->B2->A2	Server	A2	2	2	100		
A2->B3->A2	Module	B1	1	1	100		
	Module	B2	1	1	100		
ALR model: T = A + B*Length	Module	B3	1	1	100		
Coefficient A (millisecs)	0.39						
Coefficient B (millisecs/kilobyte)	0.75						
Total Number of modules and servers	4						
Maximum parallel number of modules	3						
Input: Modules/Process (0=all)	1	2	3	0	1	2	3
Input: Processes/Machine (0=all)	2	1	1	0	0	0	0
# doubles in data	100	100	100	100	100	100	100
data msg size (bytes)	808	808	808	808	808	808	808
request msg size (bytes)	4	4	4	4	4	4	4
module compute time (secs)	0.719	0.719	0.719	0.719	0.719	0.719	0.719
# iterations	11	10	10	10	11	11	10
Total time (secs) - 0.719 sec comp	24.05	25.64	21.66	30.34	23.73	24.45	21.59
Total time (secs) - 0 sec comp	0.547	0.516	0.391	0.125	0.172	0.125	0.406
Total time (secs) - 0 sec comp, 100 its				1.125			
Time/iteration (secs) - 0 sec comp	0.0497	0.0516	0.0391	0.0125	0.0156	0.0114	0.0406
Time/iteration (secs) - 0.719 sec comp	2.186	2.564	2.166	3.034	2.157	2.223	2.159
# ALR1 processes	2	1	1	1	4	2	2
# ALR2 processes	2	1	1	0	0	0	0
One Iteration Compute serial (secs)	2.876	2.876	2.876	2.876	2.876	2.876	2.876
One Iteration Compute parallel (secs)	2.157	2.157	2.157	2.157	2.157	2.157	2.157
Actual One Iteration Compute (secs)	2.186	2.564	2.166	3.034	2.157	2.223	2.159
Total No. Rem. Rqst Msgs /iteration	44	40	20	0	0	0	0
Total No. Rem. Data Msgs /iteration	44	40	20	0	0	0	0
Est. Time/Request Msg (millisecs)	0.393	0.393	0.393	0.393	0.393	0.393	0.393
Est. Time/Data Msg (millisecs)	0.996	0.996	0.996	0.996	0.996	0.996	0.996
Serial Msg Total Time (seconds)	0.061	0.056	0.028	0.000	0.000	0.000	0.000
Serial Msg Total Time/iter (seconds)	0.006	0.006	0.003	0.000	0.000	0.000	0.000
Total Serial Time/iter (seconds)	2.882	2.882	2.879	2.876	2.876	2.876	2.876
Reduction Factor	0.74	0.87	0.74	1.05	0.74	0.77	0.74

The Four Module Configuration Results (1000 bytes of data/output)

Date: 8/10/98							
System:	Type	Name	#inputs	#outputs	doubles		
A2->B1->B2->A2	Server	A2	2	2	1,000		
A2->B3->A2	Module	B1	1	1	1,000		
	Module	B2	1	1	1,000		
ALR model: T = A + B*Length	Module	B3	1	1	1,000		
Coefficient A (millisecs)	0.39						
Coefficient B (millisecs/kilobyte)	0.75						
Total Number of modules and servers	4						
Maximum parallel number of modules	3						
Input: Modules/Process (0=all)	1	2	3	0	1	2	3
Input: Processes/Machine (0=all)	2	1	1	0	0	0	0
# doubles in data	1000	1000	1000	1000	1000	1000	1000
data msg size (bytes)	8008	8008	8008	8008	8008	8008	8008
request msg size (bytes)	4	4	4	4	4	4	4
module compute time (secs)	0.719	0.719	0.719	0.719	0.719	0.719	0.719
# iterations	11	10	10	10	11	11	10
Total time (secs) - 0.719 sec comp	24.33	25.94	22.56	28.8	23.78	24.55	22.39
Total time (secs) - 0 sec comp	0.875	0.828	0.609	0.187	0.234	0.203	0.484
Total time (secs) - 0 sec comp, 100 its	5.219	5.704	4.016	1.797	1.891	1.891	4.75
Time/iteration (secs) - 0 sec comp	0.0795	0.0828	0.0609	0.0187	0.0213	0.0185	0.0484
Time/iteration (secs) - 0.719 sec comp	2.212	2.594	2.256	2.880	2.162	2.232	2.239
# ALR1 processes	2	1	1	1	4	2	2
# ALR2 processes	2	1	1	0	0	0	0
Process 1							
Modules in process	A2	A2,B2	A2,B2,B3	A2,B2,B2,B3	A2	A2,B2	A2,B2,B3
Machine	1	1	1	1	1	1	1
#total requests	11	20	10	0	0	0	0
#total data sends	22	20	10	0	0	0	0
#requests/iteration	1	2	1	0	0	0	0
#data sends/iteration	2	2	1	0	0	0	0
Process 2							
Modules in process	B2	B1,B3	B1	B2	B1,B3	B1	
Machine	1	2	2	1	1	1	
#total requests	11	20	10	0	0	0	
#total data sends	0	20	10	0	0	0	
#requests/iteration	1	2	1	0	0	0	
#data sends/iteration	0	2	1	0	0	0	
Process 3							
Modules in process	B1			B1			
Machine	2			1			
#total requests	11			0			
#total data sends	11			0			
#requests/iteration	1	0	0	0	0	0	0
#data sends/iteration	1	0	0	0	0	0	0
Process 4							
Modules in process	B3			B3			
Machine	2			1			
#total requests	11			0			
#total data sends	11			0			
#requests/iteration	1	0	0	0	0	0	0
#data sends/iteration	1	0	0	0	0	0	0

The Four Module Configuration Results (1000 bytes of data/output)

Date: 8/10/98							
System:	Type	Name	#inputs	#outputs	doubles		
A2->B1->B2->A2	Server	A2	2	2	1,000		
A2->B3->A2	Module	B1	1	1	1,000		
	Module	B2	1	1	1,000		
ALR model: T = A + B*Length	Module	B3	1	1	1,000		
Coefficient A (millisecs)	0.39						
Coefficient B (millisecs/kilobyte)	0.75						
Total Number of modules and servers	4						
Maximum parallel number of modules	3						
Input: Modules/Process (0=all)	1	2	3	0	1	2	3
Input: Processes/Machine (0=all)	2	1	1	0	0	0	0
# doubles in data	1000	1000	1000	1000	1000	1000	1000
data msg size (bytes)	8008	8008	8008	8008	8008	8008	8008
request msg size (bytes)	4	4	4	4	4	4	4
module compute time (secs)	0.719	0.719	0.719	0.719	0.719	0.719	0.719
# iterations	11	10	10	10	11	11	10
Total time (secs) - 0.719 sec comp	24.33	25.94	22.56	28.8	23.78	24.55	22.39
Total time (secs) - 0 sec comp	0.875	0.828	0.609	0.187	0.234	0.203	0.484
Total time (secs) - 0 sec comp, 100 its	5.219	5.704	4.016	1.797	1.891	1.891	4.75
Time/iteration (secs) - 0 sec comp	0.0795	0.0828	0.0609	0.0187	0.0213	0.0185	0.0484
Time/iteration (secs) - 0.719 sec comp	2.212	2.594	2.256	2.880	2.162	2.232	2.239
# ALR1 processes	2	1	1	1	4	2	2
# ALR2 processes	2	1	1	0	0	0	0
One Iteration Compute serial (secs)	2.876	2.876	2.876	2.876	2.876	2.876	2.876
One Iteration Compute parallel (secs)	2.157	2.157	2.157	2.157	2.157	2.157	2.157
Actual One Iteration Compute (secs)	2.212	2.594	2.256	2.880	2.162	2.232	2.239
Total No. Rem. Rqst Msgs /iteration	44	40	20	0	0	0	0
Total No. Rem. Data Msgs /iteration	44	40	20	0	0	0	0
Est. Time/Request Msg (millisecs)	0.393	0.393	0.393	0.393	0.393	0.393	0.393
Est. Time/Data Msg (millisecs)	6.396	6.396	6.396	6.396	6.396	6.396	6.396
Serial Msg Total Time (seconds)	0.299	0.272	0.136	0.000	0.000	0.000	0.000
Serial Msg Total Time/iter (seconds)	0.027	0.027	0.014	0.000	0.000	0.000	0.000
Total Serial Time/iter (seconds)	2.903	2.903	2.890	2.876	2.876	2.876	2.876
Reduction Factor	0.74	0.87	0.76	0.99	0.74	0.77	0.76

2.5.2 Example 2: The Eight Module Configuration Results

The Eight Module Configuration Results consists of two solvers and six modules. Each solver has two inputs and two outputs. Only one experiment is shown here, the case of outputs with 1000 doubles of data. The data table is organized the same way as the previous 4 module configuration and are shown on the following pages. The update times for the experiments were again taken to be zero seconds and 0.719 seconds. The former is all overhead and data transmission time whereas the latter has a large (compared to the overhead) update time. Looking at the time/iteration for the two cases for each experiment (column) in lines 20 and 21 shows that the overhead/data transmission time is around 76 milliseconds/iteration when there is little overlap and as low as 30 milliseconds/iteration in an experiment with large overlap.

The last line shows the reduction to 0.38 for the maximum overlap, a reduction of 2.5 : 1 in the computation time.

The Eight Module Configuration Results (1000 bytes of data/output)

Date: 8/11/98	perf8mt – 8 modules, multi-threaded to use ALR processors							
System:	Type	Name	#inputs	#outputs	doubles			
A21->B1->B2->A22->B11->B21->A21	Server	A2,A21	2	2	1,000			
A21->B3->A22->B31->A21	Module	B1,B11	1	1	1,000			
	Module	B2.B21	1	1	1,000			
ALR model: T = A + B*Length	Module	B3,B31	1	1	1,000			
Coefficient A (millisecs)	0.39							
Coefficient B (millisecs/kilobyte)	0.75							
Total Number of modules and servers	8							
Maximum parallel number of modules	3							
Input: Modules/Process (0=all)	4	3	2	1	0	1	2	3
Input: Processes/Machine (0=all)	1	2	2	4	0	0	0	0
# doubles in data	1000	1000	1000	1000	1000	1000	1000	1000
data msg size (bytes)	8008	8008	8008	8008	8008	8008	8008	8008
request msg size (bytes)	4	4	4	4	4	4	4	4
module compute time (secs)	0.719	0.719	0.719	0.719	0.719	0.719	0.719	0.719
# iterations	11	11	11	11	10	11	11	11
Total time (secs) - 0.719 sec comp	35.52	27.77	25.03	24.73	57.61	24.28	24.63	30.66
Total time (secs) - 0 sec comp	0.843	0.875	0.703	0.703	0.375	0.453	0.328	0.657
Total time (secs) - 0 sec comp, 100 its	7.719	6.719	5.406	4.875	3.567	3.171	2.813	2.516
Time/iteration (secs) - 0 sec comp	0.0766	0.0795	0.0639	0.0639	0.0375	0.0412	0.0298	0.0597
Time/iteration (secs) - 0.719 sec comp	3.229	2.525	2.275	2.248	5.761	2.207	2.239	2.787
# ALR1 processes	1	2	2	4	1	8	4	3
# ALR2 processes	1	1	2	4	0	0	0	0
Process 1								
Modules in process	A21,B21,B31,B1	A21,B21,B31	A21,B21	A21	all 8 mods	A21	A21,B21	A21,B21,B31
Machine	1	1	1	1	1	1	1	1
#total requests	22	20	11	0	0	0	11	11
#total data sends	22	20	22	11	0	0	22	22
#requests/iteration	2	1.818182	1	0	0	0	1	1
#data sends/iteration	2	1.818182	2	1	0	0	2	2
Process 2								
Modules in process	B2,B3,B11,B22	B1,B2,B3	B31,B1	B21		B21	B31,B1	B1,B2,B3
Machine	2	1	1	1		1	1	1
#total requests	22	20	11	11		0	11	11
#total data sends	22	20	0	0		0	0	0
#requests/iteration	2	1.818182	1	1	0	0	1	1
#data sends/iteration	2	1.818182	0	0	0	0	0	0
Process 3								
Modules in process		B11,A22	B2,B3	B31		B31	B2,B3	B11,A22
Machine		2	2	1		1	1	1
#total requests		20	11	11		0	11	11
#total data sends		20	11	0		0	11	11
#requests/iteration	0	1.818182	1	1	0	0	1	1
#data sends/iteration	0	1.818182	1	0	0	0	1	1
Process 4								
Modules in process			B11,A22	B1		B1	B11,A22	
Machine			2	1		1	1	
#total requests			11	0		0	11	
#total data sends			11	11		0	11	
#requests/iteration	0	0	1	0	0	0	1	0
#data sends/iteration	0	0	1	1	0	0	1	0

The Eight Module Configuration Results (1000 bytes of data/output)

Date: 8/11/98	perf8mt -- 8 modules, multi-threaded to use ALR processors							
System:	Type	Name	#inputs	#outputs	doubles			
A21->B1->B2->A22->B11->B21->A21	Server	A2,A21	2	2	1,000			
A21->B3->A22->B31->A21	Module	B1,B11	1	1	1,000			
	Module	B2,B21	1	1	1,000			
ALR model: T = A + B*Length	Module	B3,B31	1	1	1,000			
Coefficient A (millisecs)	0.39							
Coefficient B (millisecs/kilobyte)	0.75							
Total Number of modules and servers	8							
Maximum parallel number of modules	3							
Input: Modules/Process (0=all)	4	3	2	1	0	1	2	3
Input: Processes/Machine (0=all)	1	2	2	4	0	0	0	0
Process 5								
Modules in process			B2		B2			
Machine				2		1		
#total requests				11				
#total data sends				0				
#requests/iteration	0	0	0	1	0	0	0	0
#data sends/iteration	0	0	0	0	0	0	0	0
Process 6								
Modules in process			B3		B3			
Machine				2		1		
#total requests				11				
#total data sends				0				
#requests/iteration	0	0	0	1	0	0	0	0
#data sends/iteration	0	0	0	0	0	0	0	0
Process 7								
Modules in process			B11		B11			
Machine				2		1		
#total requests				0				
#total data sends				11				
#requests/iteration	0	0	0	0	0	0	0	0
#data sends/iteration	0	0	0	1	0	0	0	0
Process 8								
Modules in process			A22		A22			
Machine				2		1		
#total requests				0				
#total data sends				11				
#requests/iteration	0	0	0	0	0	0	0	0
#data sends/iteration	0	0	0	1	0	0	0	0

The Eight Module Configuration Results (1000 bytes of data/output)

Date: 8/11/98
System:
A21->B1->B2->A22->B11->B21->A21
A21->B3->A22->B31->A21

ALR model: T = A + B*Length
Coefficient A (millisecs) 0.39
Coefficient B (millisecs/kilobyte) 0.75
Total Number of modules and servers 8
Maximum parallel number of modules 3

perf8mt – 8 modules, multi-threaded to use ALR processors

Type	Name	#inputs	#outputs	doubles
Server	A2,A21	2	2	1,000
Module	B1,B11	1	1	1,000
Module	B2.B21	1	1	1,000
Module	B3,B31	1	1	1,000

Input: Modules/Process (0=all)	4	3	2	1	0	1	2	3
Input: Processes/Machine (0=all)	1	2	2	4	0	0	0	0
One Iteration Compute serial (secs)	5.752	5.752	5.752	5.752	5.752	5.752	5.752	5.752
One Iteration Compute parallel (secs)	2.157	2.157	2.157	2.157	2.157	2.157	2.157	2.157
Actual One Iteration Compute (secs)	3.229	2.525	2.275	2.248	5.761	2.207	2.239	2.787
Total No. Rem. Rqst Msgs /iteration	44	60	44	44	0	0	44	33
Total No. Rem. Data Msgs /iteration	44	60	44	44	0	0	44	33
Est. Time/Request Msg (millisecs)	0.393	0.393	0.393	0.393	0.393	0.393	0.393	0.393
Est. Time/Data Msg (millisecs)	6.396	6.396	6.396	6.396	6.396	6.396	6.396	6.396
Serial Msg Total Time (seconds)	0.299	0.407	0.299	0.299	0.000	0.000	0.299	0.224
Serial Msg Total Time/iter (seconds)	0.027	0.037	0.027	0.027	0.000	0.000	0.027	0.020
Total Serial Time/iter (seconds)	5.779	5.789	5.779	5.779	5.752	5.752	5.779	5.772
Reduction Factor	0.55	0.43	0.38	0.38	1.00	0.38	0.38	0.47

2.6 Conclusions and further work

In all the experiments, the worst case experiment is the one where all modules are in one process so that no parallel computation can be done (the fifth column of the tables where both Modules/Process and Processes/Machine are set to 0 or "all". The best case would allocate one module per process and every process to a separate machine. This could not be done with the software used for this test. The best experiment of those run was to allocate all processes to 4 machines. At any rate, the experiments indicate that the estimation of reduction of compute time can be done simply on the basis of the module update times if they are large compared to the overhead of message passing and remote-object method calls (the DCOM overhead). This is clearly the case in the experiments where a an update time of 0.719 seconds swamped the communication overhead. In fact the communication overhead was in the range of 2% to 5%. This is encouraging in that the generality in software use by basing the communication software on standard remote-object methodologies such as CORBA and DCOM does not appreciably add overhead when the module update times are high.

It is interesting to examine the time/iteration when the module update time is zero. There the overhead time varies by as much as 2.7 to 1 (and this does include the time required to transmit 1000 doubles between modules). Hence for a situation involving many modules each of which has a brief update time comparable to the overhead time can significantly reduce computation time by proper distribution across a network of processors.

The work demonstrated the desirability of using a general object methodology such as CORBA and DCOM as the basis for distributing the applications. These methodologies are heavily supported because of their importance in large commercial applications. Hence the available support for generating and distributing such objects is extensive and getting better each year.

Two interesting questions have not been resolved and could be studied in future work. The first is the detailed analysis of the potential overlap of communication overhead; the second the use of the models evaluated here as a basis for dynamic distribution of simulation applications.

Future work: Analysis of overhead overlap via queuing networks

The analysis presented in this paper can not predict the precise overlap of computation done in parallel computations. Instead, a detailed queuing network analysis must be performed to estimate this overlap. The models generated here, however, provide the necessary information needed to create such a queuing model. In those cases where the overhead is comparable to the update computation, overlap becomes very important. A study of this overlap using a queuing model would help considerably in understanding how to distribute the actual modules of a real application. Note that in the studies

reported in this paper, all the computation update-times of all modules were taken to be the same (including the update time of the solvers). This is not realistic and is a cause of great difficulty in determining the best way to distribute the application.

Future work: use of the models as basis for dynamic distribution and re-distribution of the computation

Much of the motivation for the work reported here and previous work was to allow great freedom in distributing the application to minimize the computation time. Especially important is the self-scheduling basis for the state of the individual modules developed. With the models now available, it would be possible to modify the simulation software system such that the package measures the actual time required for the update of each module during each iteration. Using these values, the model could be used to choose a better distribution for the modules to make better utilization of the available processors.

This is entirely feasible with the current design because the state of the module at any instant has been deliberately made small (iteration number, internal state number, and presence or absence of an output request for the next iteration). Hence it would be feasible to modify the program so that the moving of a module from one processor to another could be carried out as follows. First create the module in the desired processor. Second, using the same communication steps as now used at startup time, send the connector addresses to the appropriate modules connected to its inputs and outputs. Third, send the module the state information from the existing module. Finally, delete the exiting module.

This would allow the computation to review the distribution as the iterations take place and to re-distribute the modules on an individual basis at the end of an iteration. It is likely that this would be fruitful because it would seem that individual modules and sub-groups of modules might require more or less computation during the lifetime of the simulation. In particular, the number of iterations to converge a loop containing a solver might well vary over time, thereby changing the "bottleneck" in the simulation.

References

"Performance Analysis of an Actor-Based Distributed Simulation", J. D. Schoeffler, NASA report, 1997.

"A Model For Estimating Overhead in DCOM and CORBA Function Calls", J. D. Schoeffler, NASA Report, 1998