μ — A SYSTEM FOR SIMULATING AND IMPLEMENTING DISTRIBUTED AND PARALLEL ALGORITHMS

by

Miron Livny
and
Udi Manber

Computer Sciences Technical Report #737

December 1987

μ — A system for simulating and implementing distributed and parallel algorithms

(preliminary version)

Miron Livny

Department of Computer Science
University of Wisconsin
1210 W. Dayton St.
Madison, WI 53706

Udi Manber‡

Department of Computer Science
University of Arizona
Tucson, AZ 85721

November 1987

ABSTRACT

A system for simulation of distributed algorithms for a collection of workstations connected by a local area network is presented. The system, called μ, is intended for programmers who want to parallelize a program (or parts of it) in order to tap the CPU and memory space of idle workstations. Writing such parallel programs is difficult. μ simplifies this task significantly in many ways. First, μ uses only a small set of communication primitives on top of a regular programming language and makes the details of their implementation transparent to the user. μ translates the program into a discrete event system and interfaces with a simulated processing environment. The user need not be familiar at all with the simulation mechanism. Second, the environment is under complete control of the programmer. Tradeoffs such as communication speed vs. CPU speed can be easily tested, and decisions depending on them can be made more easily. Third, debugging is much simpler. The asynchronous and nondeterministic nature of the computation is preserved even though the simulation is executed on one machine. Fourth, μ is also the basis of a package for distributed programming. With it, μ programs are stripped (automatically) of their simulation parts and can be executed on several workstations with, in most cases, no modifications by the user. (This package is not completely implemented yet; it is expected to be ready very soon.) Several examples of application programs are presented.

‡ Supported in part by a National Science Foundation Presidential Young Investigator Award (grant DCR-8451397).

1. INTRODUCTION

Personal computers, workstations, and communication equipment are becoming very inexpensive. They are greatly underutilized in most cases. In a recent study of the usage of workstations in the CS department at the University of Wisconsin we have observed that on the average only 30% of the processing capacity of the workstations is utilized [ML87]. It is a challenge to use the enormous computation power of a network of computers as effectively as possible. The major portion of the work on parallel computing is devoted to "super computers" in the hope to of speeding extremely time consuming computations. In this paper we have a much more modest goal. We attempt to use the CPU and memory of idle computers in a local area network to speed up computation by a modest amount. Most of the tasks performed by a personal workstation are not very time consuming. But there are many examples of programs that require significant resources. Such examples are expected to be more common as more uses of workstations become more common. In these cases the users either wait, or they buy more powerful machines. It would be very beneficial if idle workstations could be used to their full potential.

One of the main difficulties in distributed and parallel programming is their complexities. Parallel computing is inherently much more complicated than serial computing. It is also much less understood. Many design decisions, such as how to divide the work, how to interchange data, and how to synchronize, have to be made early in the design process. They usually cannot be tested until the end, at which point it is very hard to make substantial changes. It is hard to chose among several possibilities without actually trying them. We do not yet have as good an intuition about the efficiency of parallel programs as we have about serial programs. Tools for parallel and distributed programming have only recently been developed, and a lot more work in this area is required. Of special concern from our experience is debugging. Distributed debugging is difficult since distributed programs using asynchronous communication are nondeterministic.

In this paper we describe a system called μ which simplifies the task of parallelizing programs for local area networks in several ways. The runtime environment of μ provides two modes of execution — simulation and execution. Both use the same user μ program.

Simulation

> The program is translated to a discrete event simulation program. The events correspond to either serial computation events or communication events. The serial computation parts are not merely simulated, but actually executed; the real execution time is taken as a measure of CPU consumption in the simulation environment. The communication events are simulated. We describe the method of translation in detail in the next section. The parallel execution sequence is folded to become serial. It is executed on one machine which emulates all other machines in a certain order. This order is consistent with a possible real (asynchronous) execution of the parallel program. The result is a program that very closely simulates the underlying distributed or parallel computing system. The different costs associated with communication and other details, such as the topology, are considered. This mode is used first to debug the program; it is much easier to debug on one machine and one has more control on the execution. Furthermore, since the serial timings are taken from real execution, the asynchronous nature of the distributed computation is preserved. Then, the same mode can be used to evaluate the performance of different strategies for distributing the work, synchronizing the computation, etc. The parameters of the environment can be easily changed.

Actual parallel execution

> The same program can be translated to a set of modules running on different machines connected by a local area network. The runtime environment does the communication set-up and provides support for communication primitives. This part of μ is not yet completely implemented. Consequently, we concentrate in this paper on the simulation parts.

The intended use of μ is first as a simulation, both for debugging purposes and performance evaluation, and then as a distributed program. The main advantages of μ are:

1. The user need not be aware that a simulation is being performed. The program is written in a natural convenient way, suitable for non–experts. The user can follow the execution with regular debugging statements or tracing. In this sense μ is an example of a *simulation generated automatically from a description of a system.* This issue is discussed in section 2.

2. When contemplating a conversion of a serial algorithm to a distributed algorithm, one can very easily test different strategies. The serial code can be included in the program as is. It will be executed directly by μ. Its interface to the simulator or the parallel runtime environment is automatic and transparent to the user.
3. The characteristics of the communication network and in general the environment can be separately described. Environment parameters, such as speed of communication or CPU speed, can be easily changed.

μ is not as powerful as other distributed languages. It is not meant for example to be used as a system programming language. It is intended to be used by programmers who are not experts in distributed programming or simulation who want to parallelize their programs and achieve good speedups. μ lacks some useful features, such as synchronous hand–shaking and interrupts. We made an intentional choice of introducing as few primitives as possible, making the startup time for using the system very short. Consequently, μ is very easy to use and very easy to learn. We believe that it is very suitable for its primary goal, which is to support the design and implementation of algorithms for a set of idle workstations without a heavy investment in either software or hardware. It is also very suitable for educational purposes in courses on parallel and distributed computation.

This is a preliminary report of an ongoing effort. The simulation part of the system is operational, and several experiments using it have been performed. Some of them are described in section 4. The parallel execution mode is in the final stages of implementation. Improvements and extensions of the language and its support environment are being pursued.

The organization of this report is as follows: In section 2 we discuss the basic principles behind μ's design. In section 3 we describe the language and support procedures. Section 4 contains some examples of applications, and section 5 includes further research and concluding remarks.

2. THE SIMULATION MECHANISM OF μ

A μ program contains two types of statements.

1. Regular statements of the serial programming language Modula-2, and
2. communication primitives and several other procedures supplied by μ.

Separately, there is a description of the underlying computer system that the simulation should simulate. We call this description the *environment*. A detailed description of the language is given in the next section. In this section we describe how the two types of statements are integrated in μ to form a simulation of a distributed program.

When a μ program is executed in simulation mode the parallel computation defined by the program is mapped to a discrete event system. (We assume familiarity with discrete event simulation; see for example [Zeig75].) The discrete event system that models the program is interfaced with a discrete event system that models the environment. Together they form a discrete event model of the execution of the program in the given environment. μ automatically interfaces between the two discrete event systems (see Figure 1). It passes service requests from the computation to the environment and receives completion notifications from the environment for the computation. The requests represent the computation and communication demands of the μ program. When the computing system completes the execution of a service request, it notifies the μ runtime environment. Depending on the order in which these notifications arrive, the μ interface invokes the tasks of the computation.

In order to translate a μ program into a discrete event system we have to a) identify the events of the computation, b) define the activity of each event, and c) determine the parameters that control the instant at which the event takes place. The discrete event model of the computation has to capture those aspects of the parallel computation that affect the timing of events. In the rest of this section we present the method used to dynamically perform the translation of the program. The presentation is preceded by an example of the simulated execution of a simple parallel program.

Consider Figure 2 which contains a simple example of a master-slave type of parallel computation.

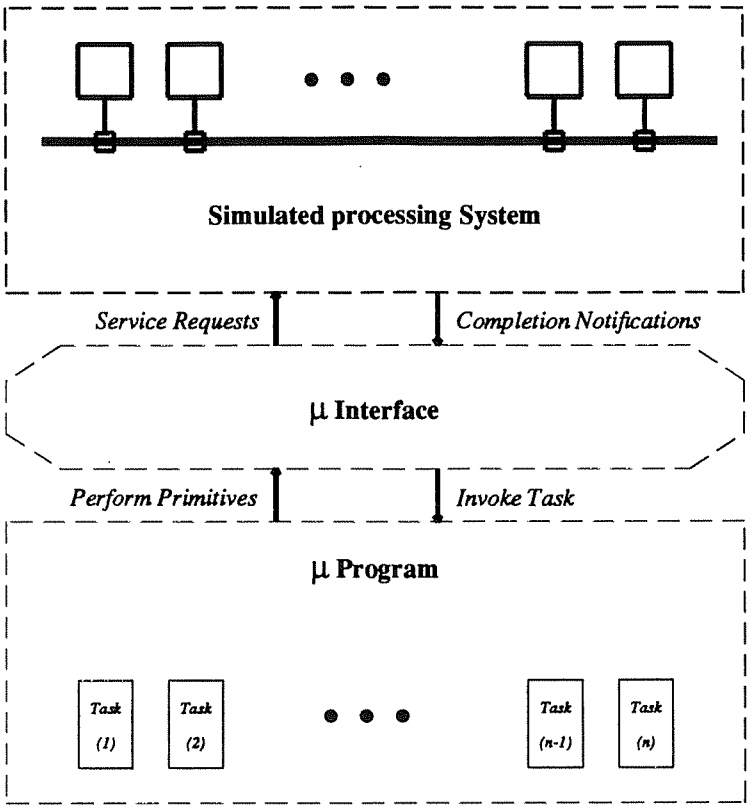

Figure 1: The Structure of the Simulation Environment

```
task Master

REPEAT
        perform local computation I ;
        SEND work to Slave ;
        perform local computation II ;
        RECEIVE results from Slave ;
UNTIL done ;
```

```
task Slave

LOOP
        RECEIVE work from Master ;
        perform local computation III ;
        SEND results to Master ;
END ;
```

Figure 2: A simple example of a parallel program

The master task may be a modified version of a regular serial program where, instead of the send and receive, the work is done locally. The slave task may be just a copy of the code for doing the work. In other words, writing the code in Figure 2 is very easy if the original serial program is already written. The hard part is evaluating what is the best way for the tasks to cooperate, and implementing the communication. μ simplifies this part.

The execution of the tasks in Figure 2 -- which can be thought of as one μ program — on a two processor system is simulated in the following way. Initially, an initiation event is scheduled for each task at time 0. Then, assuming that the master task is considered first, local computation I is executed. This is done by simply running the Modula-2 code as is, and keeping track of the CPU time it consumed. At the end of this part, which took say T_1 units of CPU time, the control returns to the μ interface. The interface

now passes a request for T_1 computation units to the simulated system and invokes the slave task. The first operation performed by the slave is a RECEIVE (for a *work* message), which blocks the task. The slave is blocked since at time 0 its input message queue is empty.

Since we assume that the master has its own CPU, its request for T_1 units of CPU will be completed at simulation time T_1. The μ interface will receive a completion notification at that simulation time. Note that it is up to the computing system to determine the allocation of CPU cycles to tasks. The completion notification indicates that the system has completed the CPU service. The μ interface now invokes the master task. The master executes the SEND *work* operation, which entails a request from the simulated environment for a message transfer. The data involved in the SEND is transferred to the slave task, and the CPU time consumed by executing the SEND is charged to the master task. The message will arrive at the slave after a delay, again simulated by the environment.

Following SEND *work* the master performs local computation II. Assuming that this computation takes T_2 CPU time units, a request for this amount of processing time is passed to the simulated system once this step of local computation is over. It may appear unreasonable to perform the computation first and only then to ask for the CPU cycles needed by the computation. In particular, computation III and SEND *results* could have been performed by the slave (in real time) before the end of computation II. This poses no problem since the master is unaware of any outside activities during computation II (or any other serial execution part). From the point of view of the discrete event simulation process there is no problem in doing so as long as RECEIVE *results*, which follows computation II, takes place after the CPU cycles were allocated to the task.

The *work* message will probably arrive at the slave sometimes during local computation II. As we said above, the master's computation is independent of the slave until RECEIVE *results* is reached. The simulation will schedule RECEIVE *results* at the appropriate simulation time (taking into account the execution time for computation II). The next event will be RECEIVE *work* by the slave, which will be triggered by the arriving message. After that, computation III will be performed (again by executing the actual code), and so on. The rest of the simulation is now clear. The simulation terminates when the master is done (exiting the loop), since at that time the slave is waiting for a message but no messages are outstanding.

The execution above depends on timings obtained by actual executions. This allows for a variance in the results since the timings will not be exactly the same every time we run the program. In addition, we can add randomness to the timings in the simulation of SEND and RECEIVE. As a result, the simulation truly models the asynchronous nature of the computation which makes debugging closer to reality. For example, if the running times for computation I and II are close then sometimes RECEIVE *results* will have to wait (block) since the results will not be there on time, and sometimes it will not wait. (It is also possible to "freeze" certain timings and make them constant if one wishes to have predictable execution sequences for debugging purposes. μ currently does not support this mode of operation, but we plan to add it eventually.)

The example above illustrates the mechanism used by μ to map a computation to a discrete event system. The mechanism is based on the following two observations:

(1) Only a small subset of the operations performed by the computation have to take place in the simulation at the same time they would have been executed in a real run of the computation. All other operations can be executed at any simulation time as long as the sequence of operations executed by a simulated task is the same as the sequence generated by a real execution.

(2) The simulation can trigger the execution of an operation even before it allocated the simulation time for it.

A simulated execution of a μ program simulates the execution of the program on a selected system. When executed by a real system each operation is performed at a certain time. We refer to this time as the *real execution instance* of the operation. The outcome of the parallel computation depends on interrelations among the real execution instances of operations performed on different machines. Fortunately, not every real execution instance must be exactly preserved by the simulation. In the example above, local computations II and III can be performed independently. They can be simulated at arbitrary times, and in particular one after the other, provided that the interactions between the two tasks are preserved. Preserving the execution instances of all operations would have caused prohibitive overheads for the simulation. It would not have been possible to simulate in reasonable time even simple parallel computations.

Consequently, we only preserve the relative timings that are important to a correct simulation. We do that in an efficient manner. We assume that message arrivals cannot interrupt a task. As a result, we have the following five operations whose timings must be preserved by the simulation:

Sending a message
: The time at which a send takes place is one of the factors that determine the arrival time of the message at its destination. Therefore, the real execution instance of a send has to be preserved by the simulation in order to guarantee that messages arrive at the same time they would have arrived in a real execution.

Message arrival
: The arrival instance of a message determines the outcome of a receive operation. All receive operations whose real execution instance occurs later than the real execution instance of a message arrival must be executed by the simulation after the arrival was simulated.

Posting a message receive request
: A task should not post a receive before all message arrival operations that may affect the outcome of the receive have been simulated.

Receiving a message
: This operation takes place when a receive is posted and a message has already arrived, or when a message arrives and the task is blocked due to a receive request. The operation can be simulated only after all message arrivals with smaller real execution instances are simulated.

Reading the clock
: When a task reads the clock (which it may want to do in order to maintain statistics) the execution instance of the operation does matter. The time value observed by the simulated task should be the same as the value observed at an actual run.

The above five operations are the only operations whose execution instance has to be preserved by the simulation. All other operations can be executed at any simulation time as long as the order of execution is preserved. If the real execution instances of OP_1 is greater than the execution instance of OP_2, the simulated execution instance of OP_2 cannot be smaller than the simulated execution instance of OP_1.

The five operations whose execution instance has to be preserved define the five events of a discrete event system that captures the behavior of a μ program. Once we have identified the events of the computation we have to decide how to break all other operations into activities. An activity is a set of operations associated with an event. The activity is executed at the time the event takes place in zero simulation time. We would like to execute all the operations that follow one event and precede the next event at the time the first event takes place. The reason for that is simple. We have to know the CPU demands of all these operations before we can schedule the latter event. The demands have to be passed to the simulated processing system that will determine when the event will take place. The only way μ can automatically determine these needs is by actually executing the operations and measuring their CPU consumption. At this point we can use our second observation, namely, that if the operation does not define an event we can perform it before we have the simulated resources to do so. We define the activity of an event as all the operations that follow the event and precede the next event.

The definition of activity given above is very dynamic. The operations that constitute an activity are determined by the computation at run time. The runtime environment of μ identifies the events of the different tasks, executes the activities, and measures the CPU usage of each activity. Once an activity has been executed, a request for service is generated and passed to the processing system. When the request has been fulfilled by the system the μ runtime is notified and the task that generated the given service request is activated. The task performs the activity of its next event and returns control to the μ interface when the next event is reached. The CPU consumption of the activity is determined, and a new service request is passed to the processing system.

3. THE μ LANGUAGE

μ is a parallel programming language based on Modula-2. It views a parallel computation as a fully connected set of processes. Each process is an instance of a μ task, and is realized by a Modula-2 process. A process has a unique identification number and can exchange messages directly with all other processes. A

task is an independent compilation unit represented by an *implementation module* and an optional *definition module*. The logic and data structures of the task are captured by the implementation module. The definition module is used for exporting type declaration across task boundaries. μ modules are compiled into Modula-2 modules, which are then compiled into object code.

A parallel computation is constructed from a given set of μ tasks by means of the *environment* file. The total number of tasks as well as the number of instances of a particular task may vary from one computation to the other. Information stored in the file specifies how many tasks are included in the computation, their type, and the values of their input parameters. Input parameters can determine the behavior of a task and thus be used to personalize different instances of the same task. For instance, if we consider the example of Figure 2, the environment file will determine the number of slaves in the computation as well as the identification number of the master and all slaves. The environment file will also include for each instance of a slave a value for an input parameter that that controls the amount of work performed after a message is received (local computation III). The construction of the computation is dynamic and takes place at run time. The runtime environment of μ interpret the file, creates the a process for every instance of a task, and assigns values to the input parameters of each instance. Note that many different computations can be built from one executable.

Every two processes that are part of the same computation can exchange messages. The runtime environment of μ establishes communication links to support a fully connected communication scheme. A μ message consists of a fixed header and a variable size data record (see Figure 3). The header has five fields — source, destination, type, value, and data. The first two fields store the identification numbers of the source and the destination tasks. The type field identifies the type of message. Message types are defined by the user and are used to simplify programming. It enables the recipient to select messages conveniently. The data field is a pointer to a user defined record. There is no limit on the size of this record. The value field is used for short messages. One word (4 bytes) is allocated for this purpose. It saves the programmer the need to define the data record.

μ has a single send primitive. The syntax of the send statement is the following:

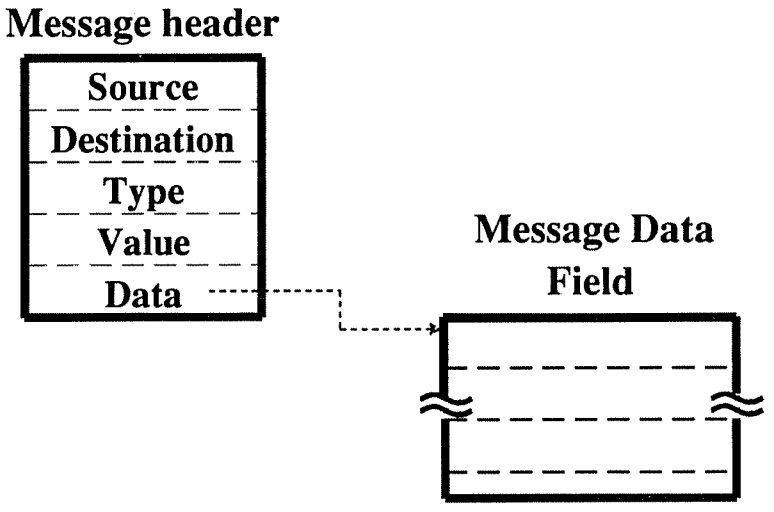

Figure 3: Structure of a μ Message

```
SendStatement    =   "SEND(" { SendOptionList } ")"
SendOptionList   =   SendOption { "," SendOptionList }
SendOption       =   "Dest" "=" TaskIdList | "MsgType" "=" Identifier |
                     "Value" "=" Expression | "Data" "=" DataAddress
TaskIdList       =   Expression { "," Expression }
DataAddress      =   Identifier | "ADR" "(" Identifier ")"
```

The send primitive has four fields - Dest, MsgType, Value, and Data. These fields correspond to the last four fields of the header. All four send fields are optional. The default value of the Dest field is 0, which corresponds to a broadcast. In other words, by not including a destination number the message becomes a broadcast. The default value of the MsgType and Data fields is null. A null message type matches all types. A null data pointer implies that the data is stored in the value field. The default value for the value field is zero.

The send operation of μ is a non–blocking operation. When a send takes place a header record is created and the content of the data field (if present) is copied to a buffer provided by μ. The original copy of the data remains untouched. When more than one task identification number is given, a message (a header and data records) is created for each task listed. The expression assigned to the value field has to be of type INTEGER. Type casting is used when a non INTEGER value is stored in the value field.

The receive primitive is a blocking receive controlled by a timeout mechanism. Its syntax is as follows.

```
ReceiveStatement    =   "RECEIVE(" { ReceiveOptionList } ")"
ReceiveOptionList   =   ReceiveOption { "," ReceiveOption }
ReceiveOption       =   "Source" "=" TaskIdList | "MsgType" "=" Identifier |
                        "Value" "=" Identifier | "Data" "=" DataAddress | "TimeOut" "=" Expression
```

The receive has five fields. The first two fields are used to select the source and the type of the message to be received. The task waits until a message with the given source and type arrives, or until the timeout expires. The data record of the message received is copied by μ to the memory location defined by the Data field. The Data field can be viewed as a buffer for incoming messages provided by the task. The Value field is used when no Data field is given in the corresponding SEND.

All receive fields, like send fields, are optional. The default value of the MsgType is null, which means that no selection is performed. If the Source field is not specified then messages from any source will be acceptable. If the Data field is not specified then the *value* is used (the default value is 0). Either Data or Value must be provided. The default value of the TimeOut field is infinite. When the TimeOut field is assigned a finite value the receive is aborted if no matching message arrives before the timeout expires. A zero timeout value turns the receive into a non–blocking receive. The address of the header record of the received message is stored in a predefined local variable named *CurrentMessage*. Before a new value is assigned to CurrentMessage its value is checked and if it points to a valid header record the record is disposed. By doing so, μ disposes the header of the last message received by a task when a new message is received unless a specific action is taken by the programmer to keep the header. In such a case it is the programmers responsibility to free the space used by the header once the header is not needed any more.

To support peeking at messages queued in μ buffers we include an operation, CheckMessages, that counts the number of matching messages currently in the task input buffer. All messages that have arrived at the processor that serves the task but have not been received yet are kept in the task input queue. CheckMessages returns an integer value which is a count of the number of messages in the queue that have a source and type fields as defined by the two fields of the operation. The syntax of CheckMessages is given below.

```
CheckStatement    =   "CHECKMESSAGES(" { CheckOptionList } ")"
CheckOptionList   =   CheckOption { "," CheckOptionList }
CheckOption       =   "Source" "=" TaskIdList | "MsgType" "=" Identifier
```

The two CheckMessage fields have the same default values as the corresponding receive fields. CheckMessages can help in situations where only the number of incoming messages is relevant and it is not

desired to remove them from the queue yet.

Since the meaning of time depends on whether the program is executed on a real or simulated system, reading the clock has to be performed via μ. A μ task should not read the processor timer directly. The procedure **GetTime (VAR TimeVal : REAL)** is used for timing. In case of a simulated execution it returns the value of the simulation clock at the time the procedure was executed. When the computation runs on a real system μ will read the hardware clock of the processor executing the task. The procedure **ConsumeProcessor(Quantity : REAL)** generates a demand for *Quantity* units of processing units. When the program runs on a real system the demand is translated to a sequence of instructions executed on behalf of the task. The number of instructions depends on the characteristics of the processor that executes the task. When the program is simulated the demand is translated to a service request that is passed to the simulated system. ConsumeProcessor does not actually use the corresponding CPU units in the case of simulated execution, thus it is much more efficient and should be used whenever possible. It can be used to replace parts of the algorithm whose running times are known or can be approximated sufficiently and their result is known too.

The design of μ is still not cast in iron. We expect to add several more tools and library routines as more applications programs are written.

4. EXAMPLES OF APPLICATIONS

In this section we briefly describe four application programs that have been written for μ or are currently in development. Describing the algorithms in detail is beyond the scope of this paper; we only discuss the main issues related to μ. Our purpose is to illustrate the use of μ, and show its diversity. As was mentioned in the introduction, we see μ as a tool for designing and evaluating parallel algorithms. Hence, our first experiments were to evaluate algorithms that we are currently studying and researching. μ was very helpful in this respect. It pointed out to us several aspects of the algorithms which we have not considered before. In particular, μ can facilitate the study of efficiency of parallel algorithms as a function of the speed of communication. Most parallel algorithm are designed either for very fast parallel computers with very fast communication, or for distributed systems with very slow communication. We believe that there is a wide "gray area" in between. It is important to see, for example, the effects of faster networks (or slower networks) on efficiency of certain applications. These effects are currently not well understood. For example, we encountered one anomaly, described below, in which a slower network led to a faster execution.

4.1. Petri net analysis

The first example involves a parallel algorithm for petri net analysis [LMV88]. The main part of the algorithm involves building a graph (of all the Petri net states). This is done essentially by starting with initial known nodes (states) in the graph, and generating new nodes from them. Since the graph is not necessarily a tree or even acyclic, the same node may be generated from several different nodes. Hence, one needs to verify, when a node is generated, whether it is a new or an old node. This is done with the use of shared memory (which contains the identity of all previously generated nodes). The shared memory version has been implemented on a Sequent parallel computer with very good results. We want to parallelize this algorithm without the use of shared memory.

This work typifies the use of μ. We started with a parallel algorithm, developed for a shared memory. The algorithm uses the shared memory very heavily, and it seems initially that having a fast shared memory is required in order to get any speedup in the computation. We developed a distributed algorithm, based on the original algorithm, but with different strategies for achieving parallelism. The performance of the algorithm depends on several parameters that have to be "tuned." With μ it was easy first to convert the parallel algorithm to a distributed one, and then to test all the parameters and see their effect.

We found, for example, that the speedup was less dependent on the delay on the channel, and more dependent on the CPU requirements of sending and receiving a message. We were surprised to find that reasonable speedups are achievable even with slow communication. For example, with 8 processors, communication delays of 0.01 seconds per message, and CPU overheads of 0.001 seconds per send and

receive, the speedup for a large petri net was about 6.5. At some point, we observed that delaying a message further on the channel improves the speedup! This apparent paradox results from the load balancing mechanism in the algorithm. When a processor is "out of work" it sends messages requesting work from other processors. When the computation is near the end, the overhead generated from the load balancing only interferes with the computation, and it is better to stop it. But, it is very hard to detect that the end of the computation is near. If the delay is higher, less request messages pass through and less overhead is generated! Hence, the overall performance is improved. This affects only a small part of the algorithm, but it is a very good example of the type of observations that can be easily learned by using μ.

4.2. Disseminating information in a local area network

The second example involves algorithms to disseminate information in a local area network developed by Alon, Barak and Manber [ABM86]. The key feature of these algorithms is that information is disseminated quickly and reliably without the use of broadcast. Broadcasts are expensive since they force all stations to receive messages even if not all of them are "interested". The cost of receiving a message is a substantial part of the cost of communication. The algorithms presented in [ABM86] use slightly more messages than the straightforward broadcast based algorithms, but substantially fewer receive operations. Overall, they give less delay and less CPU usage. It is not easy to extract the cost of receiving messages from experimental results. It is easy, however, to extract it (or any other partial cost) from the simulations. We have implemented one variant of these algorithms on a collection of MicroVaxes II connected by an Ethernet. This implementation took several weeks and it is quite involved mainly because of the communication protocol and set-up. The simulation of the algorithm in μ took 2 days. The comparison of effort is not entirely fair since the first implementation was intended for "production" use and it includes quite a bit of fine tuning. However, evaluating the algorithms and their main ideas can be done just as well with μ. The μ simulations verified the theoretical predictions about the performance of these algorithms.

4.3. DIB — a distributed implementation of backtracking

The third example is a complete translation of the DIB package for distributed backtracking developed by Finkel and Manber [FM87]. DIB, which is a rather large and complicated distributed program implemented in the Crystal multiprocessor system, utilizes the special properties and form of backtracking algorithms. DIB takes backtracking application programs without any parallel programming statements and automatically creates distributed programs from them. Thus the user need not worry about parallelism or any of the programming problems associated with it. On the other hand, the development of DIB was very difficult due to the complexity of implementing distributed algorithms. In particular, debugging was extremely difficult. It depended on the many layers of protocols and systems that DIB used (not all of which were bug–free).

The translation of DIB to μ would greatly simplify the continued development of DIB and its extensions. It would also simplify debugging of application programs. When the distributed implementation of μ is completed, DIB will run on any set of machines running UNIX and connected through the TCP/IP protocol (whereas now it requires the special communication primitives of the Crystal multicomputer).

4.4. Distributed algorithms for minimum cost spanning trees

The fourth example is algorithms for distributed computation and maintenance of minimum cost spanning trees. Given a connected graph with costs associated with the edges, a minimum cost spanning tree (MCST) is a connected subgraph containing all the vertices such that the total costs of all its edges is minimized. If the graph corresponds to a point–to–point computer network (such as the Arpanet) the MCST is very useful for broadcasts. A broadcast delivered only across the MCST edges incurs minimal cost. Several distributed algorithms were suggested for computing MCST ([GHP83]). There are also algorithms for updating MCST under certain changes [Ch85]. We are evaluating these algorithm using μ. This example is different from the two above in that the algorithms are not intended to run on a local area (fully connected) network. μ assumes that all nodes can communicate with all others, but the algorithm

under simulation does not need to use all connections of course. Even though μ was designed for evaluating and developing algorithms for local area networks, one can impose a topology on the algorithms. The simulated processing system can capture the properties of a point-to-point network, however in many cases, and in many phases of the design, it is easier to ignore the topology.

5. FURTHER RESEARCH AND CONCLUSIONS

We have presented a system for design and evaluation of distributed algorithms for local area networks. We are currently extending the system to more types of parallel algorithms. First, we plan to add shared memory access primitives and locking primitives. The underlying framework will remain the same for shared memory algorithms. That is, debugging and performance evaluation can be done in a simulated environment using the same interface to the simulator. Actual translation to a shared memory system will be harder, and will probably depend more heavily on the specific system. However, it can still serve as a tool for comparing tightly coupled and loosely coupled approaches to parallel algorithms. Second, we plan to improve the user interface. Adding flexible reporting and tracing, and providing graphic animation will be extremely helpful for debugging. Third, we plan to add library routines to simplify coding of common types of distributed algorithms. For example, programs for matrix calculation, graph manipulation, and other distributed data structures are under development. Fourth, we plan to build a library of environments including many commonly used computers and networks. We intent to use DeNet [Liv87], which is a Modula-2 based discrete event simulation language, to construct the simulators of different systems. In order to achieve this goal we plan to build an interface between μ and DeNet and to integrate μ into the DeLab simulation laboratory. Such an integration would simplify many experiments. In particular, it will allow testing of parallel programs under different environments to determine the effects of key parameters on the efficiency of the programs.

Acknowledgements

P. Lai, L. McVoy, B. Narendran, V. Srinivasan, and M. Subramanyan implemented parts of the software and helped with the design. Greg Andrews made many helpful comments that improved the presentation.

REFERENCES

[ABM86]
N. Alon, A. Barak, and U. Manber, "On Disseminating Information Reliably without Broadcasting," *the 7th International Conference on Distributed Computing Systems*, Berlin, (September 1987).

[Ch85]
H. H. Chen, "Incremental Computations of Topological Properties in Distributed Networks," Ph.D. thesis, University of Wisconsin — Madison, October 1985.

[FM87]
R.A. Finkel and U. Manber, "DIB — A Distributed Implementation of Backtracking," *ACM Transactions on Programming Languages and Systems*, (April 1987), pp. 235-256.

[GHP83]
R.G. Gallager, P.A. Humblet and P.M. Spira, "A Distributed Algorithm for Minimum-Weight Spanning Trees," *ACM Trans. on Programming Languages and Systems*, (January 1983), pp.66-77.

[Liv87]
"DeLab - A Simulation Laboratory," to be presented at " the 1987 Winter Simulation Conference," Atlanta, Georgia, December 1987.

[LMV88]
P. Lai, U. Manber, and M. Vernon, "Converting a shared memory algorithm to a distributed algorithm — a case study," in preparation.

[ML87]
M. Mutka and M. Livny, "Profiling Workstations' Available Capacity For Remote Execution," to be presented at "PERFORMANCE '87, the 12th IFIP W.G. 7.3 International Symposium on Computer Performance Modeling, Measurement and Evaluation," Brussels, Belgium, December 1987.

[Zeig76]
B. P. Zeigler, "Theory of Modeling and Simulation," John Wiley, New York, 1976.

EFFICIENT ALGORITHMS FOR POLYHEDRON COLLISION DETECTION

by

Deborah A. Joseph

and

W. Harry Plantinga

Computer Sciences Technical Report #738

December 1987

Efficient Algorithms for Polyhedron Collision Detection*

Deborah A. Joseph – W. Harry Plantinga

Department of Computer Sciences
University of Wisconsin - Madison

Abstract

In this paper we present efficient algorithms for determining whether polyhedra undergoing linear translations will collide. We present algorithms for a finding collisions between a pair of moving convex polyhedra and among several moving non-convex polyhedra. With the non-convex algorithm we also show how to determine *when* the objects will collide. The algorithms work by considering time to be a fourth dimension and solving the related 4-D polytope separation problem. In the convex case we present an algorithm that runs in O(**n**) time where **n** is the sum of the sizes of the objects. The algorithm is a generalization of an O(**n**)-time polyhedron separation algorithm of Dobkin and Kirkpatrick to 4-D convex prisms. In the non-convex case we present an algorithm that runs in $O(\mathbf{n}^2)$-time. The algorithm is a generalization of the naive polyhedron separation algorithm. We generalize the naive algorithm to finding the separation of d-polytopes in E^d for any d, although we use only the 4-D case for collision detection. Both algorithms use O(**n**) space. In the process of finding the separation in the non-convex case we must determine whether a point is in a polytope in E^d, and we present an algorithm for doing that.

* This work was supported in part by the National Science Foundation under grants DCR-8520870 and DCR-8402375 and a faculty development grant from AT&T.

1. Introduction

Determining whether moving objects will collide is important for applications that involve the simulation of physical systems. For example, we may want to determine whether the parts of an automobile engine can move freely or whether moving objects such as airplanes will collide. We can determine these things with a collision detection algorithm. Collision detection is also important for motion planning, since the motions of an object must not cause it to collide with any obstacles. In robotics, task-level languages for describing motions of a robot must include a collision detection algorithm if they are to prevent motions that would cause the robot to collide with an obstacle. In addition to knowing *whether* objects will collide, we may want to know *when* and *where* they collide in order to modify the engine part, alert the pilot, or stop the motion of the robot at an appropriate time.

Previous work on collision detection has taken three main directions. For a single object moving among stationary obstacles, it is sufficient to determine the volume of space that the object sweeps out as it moves. The object collides with an obstacle if the swept volume intersects the obstacle. Boyse (1979) introduces this method, dealing separately with the cases of object rotation and translation, but he does not analyze the time complexity of the algorithm. Ganter (1985) constructs an approximation to the swept volume by constructing a "skin" over a finite number of copies of the object at points along its path. However, neither of these algorithms determines the time of collision, and no algorithm using this method can handle the case where more than one object is moving. If more than one object is moving, such an algorithm will report an intersection even if two objects pass over the same point at different times.

Another direction that researchers have taken is to simulate the motion of an object or objects by checking for intersections at closely-spaced points in time. Examples include work by Meyer (1981), Cameron (1985), Culley and Kempf (1986), and Hayward (1986). One disadvantage of this approach is that it is time-consuming to do so many intersection tests. Another disadvantage is that it is not completely accurate—the algorithm may not detect collisions that occur between intersections tests if special precautions are not taken in the choice of time intervals.

The third approach is that of solving the problem in configuration space (Lozano-Perez, 1983). A configuration-space algorithm is one in which a point in the configuration space represents the configuration of the problem—for example, the

location and orientation of the moving object. Obstacles in configuration space correspond to regions of forbidden configurations—that is, configurations in which the object intersects an obstacle. Canny (1986) uses a configuration-space algorithm to find collisions among obstacles and objects that may be translating and rotating. The algorithm runs in $O(\mathbf{n}^2 \log \mathbf{n})$ time where $\mathbf{n}$ is the sum of the sizes of the object and obstacles. However, in order to work within that time bound his algorithm is limited to a collection of convex objects and obstacles. Therefore, he assumes that objects and obstacles are given as a collection of possibly overlapping convex pieces. Chazelle (1984) has shown that decomposing a non-convex polyhedron can require $\Omega(\mathbf{n}^2)$ convex pieces. Thus, Canny's algorithm may require time $\Omega(\mathbf{n}^4 \log \mathbf{n})$ for general polyhedra.

In this paper we take an approach proposed by Cameron (1985). We consider time to be a fourth dimension and determine the volume of *space-time* that the moving objects occupy. The collision detection problem then becomes a 4-dimensional intersection detection problem. Although Cameron proposes the idea as one of three possible approaches to the problem, he does not present an algorithm for this approach. We present algorithms using this approach and analyze their complexity.

Our approach differs from the configuration-space approach in that a point of configuration space represents the location and orientation of the object while a point of space-time represents a point of space at a particular time. Solving the problem in configuration space reduces the problem to one of determining whether a one-dimensional manifold of points intersects a volume of configuration space, but the configuration space is 6-dimensional, and it is difficult to construct the constraints for non-convex moving objects. Solving the problem in object space enables us to work in a 3-dimensional space, but the problem is complicated by the fact that the objects are moving. Our approach of solving the problem in space-time is in a sense a compromise: the space is 4-dimensional and the problem to be solved is a static intersection problem.

We restrict our attention to polyhedral objects undergoing linear translation, and within this setting we deal with two cases: two moving convex objects and several moving non-convex objects. We present an $O(\mathbf{n})$-time algorithm for the former case and an $O(\mathbf{n}^2)$-time algorithm for the latter case. Both algorithms use $O(\mathbf{n})$ space. The algorithm for the convex case is a generalization of the polyhedron separation algorithm of Dobkin and Kirkpatrick (1983) to convex 4-prisms. The algorithm for the general case is a generalization of the naive algorithm for finding the separation of polyhedra, by finding and minimizing the separation of every pair of faces. We generalize this algorithm to finding the separation of polytopes in E^d (d-

dimensional Euclidean space), although for the collision detection algorithm we make use only of the 4-dimensional case. We also show how to determine the time location of the first collision of the objects.

Higher-dimensional cases of the polytope separation algorithm may prove useful in algorithms for various problems using a configuration-space approach. The polytope separation algorithm uses two auxiliary algorithms, one for determining whether a point is in a d-polytope in E^d and one for determining whether a line segment intersects a $(d-1)$-polytope in E^d. We present mutually recursive algorithms for these problems that run in time linear in the number of incidences of the polytope (as defined below). These algorithms are of interest in themselves.

In Section 2 we present the algorithm for the convex case. In Section 3 we present the algorithm for the general case. In Section 3.1 we discuss the representation and size of polytopes. Section 3.2 contains the algorithm for finding the separation of polytopes in E^d and discusses using the polytope separation algorithm for polyhedron collision detection. In Sections 3.3 and 3.4 we present two auxiliary algorithms that are used by the separation algorithm. In Section 3.5 we show that we can use the results of the previous sections to find collisions among moving general polyhedra in $O(\mathbf{n}^2)$ time. In Section 4 we discuss extending the results of Section 3 to objects rotating and undergoing other nonlinear motion.

2. Determining Whether Two Moving Convex Polyhedra will Collide

Suppose a convex polyhedron **P** undergoes linear motion, that is, motion along a line at a constant speed. If we consider time to be a dimension, then for any translation along a line segment there is a corresponding convex 4-prism with basis **P** in space-time representing that motion. The ends of the 4-prism are copies of the original object at the initial and final times and locations. Thus, for a translation along the vector (d_x, d_y, d_z) taking time d_t, the 4-prism is the Minkowski sum of the polyhedron at the initial time and location in E^4 and the vector (d_x, d_y, d_z, d_t). The problem of determining whether the two moving convex polyhedra will collide becomes a 4-dimensional intersection problem: the related 4-prisms share a point in space-time if and only if the objects collide.

There are no linear-time algorithms for separation of polytopes of dimension 4 or higher. However, Dobkin and Kirkpatrick (1983) present a linear-time algorithm for determining the separation of two convex polyhedra in E^3 that generalizes to finding the separation of convex 4-prisms. The Dobkin-Kirkpatrick algorithm involves two steps: constructing a hierarchical representation of the polyhedra and

finding the separation of the polyhedra at progressively more detailed levels of the hierarchy. The only part of the algorithm that is specialized to three dimensions or less is the construction of the hierarchical representation, and that is done because in higher dimension it is not necessarily of linear size. However, we will show that the size is linear for convex 4-prisms.

The algorithm for constructing the 4-prisms representing the motion of a polyhedron **P** from times t_0 to t_1 is the following: begin with two copies of **P**, one at the initial location and one at the location after the translation. Add to each vertex of the copy of **P** at its initial location a fourth coordinate, t_0, representing the initial time. Add the time coordinate t_1 to each vertex of the copy of **P** at its final location. Then connect each corresponding pair of vertices of the two ends of the prism with an edge. Connect corresponding pairs of edges with a 2-dimensional face or 2-face (in fact, a parallelogram). Finally, connect corresponding pairs of faces with a 3-face (a 3-prism).

The hierarchical representation for 4-prisms is constructed in the same manner that Dobkin and Kirkpatrick construct the representation for polyhedra. The hierarchical representation is a sequence of approximations to the shape of the polyhedron, each less detailed (i.e. with fewer of the vertices of **P**) than the last. Each successive approximation of the shape is formed by removing an independent set of vertices from the current approximation of the shape and taking the convex hull of the remaining vertices. Formally, the hierarchical representation is defined in the following way: if **P** is a d-polytope with vertex set V(**P**), a sequence of polytopes H(**P**) = $\mathbf{P}_1, \ldots, \mathbf{P}_k$ is said to be a *hierarchical representation* of **P** if

(1) $\mathbf{P}_1 = \mathbf{P}$ and $\mathbf{P}_k$ is a d-simplex;
(2) $\mathbf{P}_{i+1} \subseteq \mathbf{P}_i$ for $1 \leq i < k$;
(3) $V(\mathbf{P}_{i+1}) \subseteq V(\mathbf{P}_i)$; and
(4) The vertices of $V(\mathbf{P}_i) - V(\mathbf{P}_{i+1})$ form an independent set (i.e. are non-adjacent) in $\mathbf{P}_i$.

The *height* of H(**P**) is k and the size is the sum of the sizes of the levels. The *degree* of H(**P**) is the maximum degree of any vertex removed at any level.

Dobkin and Kirkpatrick prove that a hierarchical representation for a polyhedron has height O(log **n**) and size O(**n**) where **n** is the size of the polyhedron, if the vertices removed to form each successive level of the hierarchy form a maximal independent set of the vertices of degree at most b for some b. The algorithm for constructing such a representation is to repeatedly remove a maximal independent set of the vertices of degree at most b from the polyhedron (with the next level of the

hierarchical representation consisting of the convex hull of the remaining vertices) until all that remains is a tetrahedron.

For polytopes of dimension greater than 3, the hierarchical representation has maximum size greater than $O(\mathbf{n})$, however. The reason is that the skeleton (or vertex-edge graph) for a 4-polytope is not necessarily planar, so that vertices may be adjacent to more of the other vertices in the skeleton. As a result, the maximal independent sets may be smaller than they are for planar graphs, so that the number of vertices removed at each level of the hierarchical representation may be smaller, resulting in a larger representation. In fact, for the "cyclic" polytopes in E^4 (defined as the convex hull of the points $(\mathbf{n}, \mathbf{n}^2, \mathbf{n}^3, \mathbf{n}^4)$ for $\mathbf{n} \geq 5$ (Grünbaum, 1967)), every pair of vertices is connected by an edge, so the skeleton is a complete graph. Thus only one vertex is removed at each level of the hierarchical representation. The size of a cyclic polytope is $\Theta(\mathbf{n}^2)$ and the representation has $\Theta(\mathbf{n})$ levels, so its size is $\Theta(\mathbf{n}^3)$.

The skeleton for a convex 4-prism is not in general planar. However, since both ends of the prism are convex polyhedra and the remaining faces are in one-to-one correspondence with the faces of one of the ends, the skeleton for a convex prism is not much more complex than that of the convex polyhedron. We construct the hierarchy for the 4-prism in the same manner: repeatedly choose and remove maximal independent sets of the vertices of degree at most b. The following lemma (which is a modification of Dobkin and Kirkpatrick's Lemma 3.1) shows that we can construct a hierarchical representation for 4-prisms that has height $O(\log \mathbf{n})$ and size $O(\mathbf{n})$ where $\mathbf{n}$ is the size of the 4-prism.

Lemma 1. There exist constants $b > 1$ and $c < 1$ such that for all convex 4-prisms $\mathbf{P}$, the algorithm above produces a hierarchical representation of $\mathbf{P}$, $H(\mathbf{P}) = \mathbf{P}_1, \ldots, \mathbf{P}_k$, with degree at most b such that $|\mathbf{P}_{i+1}| < c\,|\mathbf{P}_i|$, $1 \leq i < k$.

Proof. First we show that the lemma is true for polyhedra; the result for 4-prisms will follow. The skeleton of any polyhedron $\mathbf{Q}$ is connected and planar. Let e be the number of edges, v the number of vertices, and k the number of vertices of degree $>b$. Every vertex has degree at least 3, so the number of edges satisfies

$$e \geq (3v + (b-3)\,k)/2$$

Each element of the maximal independent set can cover at most itself and b other vertices, so

$$(1-c)\,v \geq (v\,k)/(b+1)$$

Together with a choice of $b = 11$, these imply that $e \geq v\,(48\,c - 85/2)$. But by Euler's formula, any connected planar graph has $e \leq 3\,v - 6$. Thus for a choice of $b = 11$, the consequence of the lemma is satisfied for polyhedra when $c = 19/20$.

P is a 4-prism, not a polyhedron. However, the skeleton of **P** consists of two copies of the skeleton of the ends of **P**, say $\mathbf{S}_1$ and $\mathbf{S}_2$, with corresponding vertices connected. All of the vertices of **P** are in $\mathbf{S}_1$ or $\mathbf{S}_2$, so half of them are in $\mathbf{S}_1$. Since a maximal independent set of vertices of $\mathbf{S}_1$ is an independent set in **P**, the constants $b = 11$ and $c = 39/40$ satisfy the conditions of the lemma for 4-prisms. ■

Thus the algorithm above produces a hierarchical representation for an arbitrary convex 4-prism **P** with degree at most b, height $O(\log(|\mathbf{P}|))$, and size $O(|\mathbf{P}|)$. This hierarchical representation can be constructed in linear time in the same manner that Dobkin and Kirkpatrick's algorithm constructs the representation for polyhedra. In practice, the fraction of vertices removed at each level of the hierarchy will usually be much larger than 1/40.

The remainder of Dobkin and Kirkpatrick's algorithm is not specialized to two and three dimensions and works also for 4-polytopes. Thus, the algorithm for determining whether two convex polytopes **P** and **Q** undergoing linear translation will collide proceeds as presented below. In the third step, the closest pair is found in linear time using the method of Dobkin and Kirkpatrick.

Construct the 4-prisms corresponding to the motions of **P** and **Q**. Call them **P'** and **Q'**.

Construct the hierarchical representations $\mathbf{P}_1, \ldots, \mathbf{P}_r$ for **P'** and $\mathbf{Q}_1, \ldots, \mathbf{Q}_s$ for **Q'**.

$i \leftarrow \min(r,s)$

$(\mathbf{p},\mathbf{q}) \leftarrow$ closest_pair($\mathbf{P}_i$,$\mathbf{Q}_i$)

while $\mathbf{p} \neq \mathbf{q}$ and $i > 1$ **do begin**

$i \leftarrow i - 1$

$(\mathbf{p},\mathbf{q}) \leftarrow$ closest_pair($\mathbf{P}_i$,$\mathbf{Q}_i$) **end**

separation(**P**,**Q**) $\leftarrow$ |**p**-**q**|

Algorithm 1. Determining whether convex polyhedra will collide.

3. Determining Whether Several General Polyhedra will Collide

We can transform the problem of finding collisions among several translating objects into a four-dimensional intersection problem in the same manner as the convex case. However, the 4-polytopes related to the motions are general 4-prisms rather than convex 4-prisms. Thus, in this case we cannot generalize an algorithm designed for convex objects, but we can generalize the naive algorithm for finding the separation of two polyhedra. To find the separation of several polyhedra we find the separation of every pair and minimize.

The naive algorithm for finding the separation of two polyhedra finds the separation of every pair of faces of the polyhedra and minimizes. If the minimum separation is zero, the objects intersect; if it is greater than zero, it is also necessary to test whether one object has a point inside the other. If so, then one object is completely inside the other; if not, then the objects do not intersect. This algorithm generalizes in a straightforward way to E^d, except that it is also necessary to generalize other auxiliary algorithms that are relatively easy in E^3. These algorithms determine (1) whether a point is in a d-polytope in E^d and (2) whether a line segment intersects a $(d-1)$-polytope in E^d. We call these the point–polytope and segment–facet algorithms and present them below.

We find the time of intersection by noting that at the instant that the objects first collide, even though they may collide in many points at once, some pair of faces intersects in a single point. (We prove this below.) Thus to find the time of collision it suffices to find every one-point intersection of a pair of faces and find the one that occurs first. The location of the point is a point of the collision.

In this paper we will call a $(d-1)$-dimensional affine subspace of E^d a *hyperplane*. Functionally, it is the set of points $\{ \mathbf{x} \mid \mathbf{x}\cdot\mathbf{u} = a \}$, where $\mathbf{u}$ is a direction and a is a constant. We call the intersection of a polytope with a supporting hyperplane a *face*, and we call a d-dimensional face a *d-face*. We call a (d–1)-face a *facet*. The affine subspace spanned by a face $\mathbf{f}$, denoted Aff($\mathbf{f}$), we call a *flat*. Thus, a flat of dimension $d-1$ in E^d is a hyperplane.

We say that two flats are *partly parallel* if they are linearly dependent; in that case, when translated to the origin their intersection is more than a single point. Thus, for example, two planes in E^3 are always partly parallel. Otherwise the flats have a unique mutual normal line (unique in the smallest affine space containing both flats) and are *skew* or intersect in a single point. We call the intersection points of this normal line with the flats the *feet* of the normal. We will also speak of a pair of faces of polytopes as having a mutual normal; in this case we refer to the normal between the flats spanned by the faces. We say that a line segment $\mathbf{s}$ *realizes* the

separation of two polytopes when it is a minimal-length line segment that meets both polytopes. In the case of two skew lines in E^3, the unique mutual normal is the line containing the shortest line segment joining them, and the feet of the normal are the intersection points. The line segment joining them realizes the separation.

3.1. The Representation and Size of Polytopes

Since this algorithm is defined for polytopes in E^d for any d, we first discuss ways of representing polytopes and the size of such representations. There are at least three different measures of the size of a polytope that at first seem reasonable. All of these measures are asymptotically within a constant factor of each other for polygons and polyhedra, but they differ for polytopes of dimension ≥ 4. The first measure is only reasonable for convex polytopes: the number of vertices. In the convex case, a list of vertices is sufficient to reconstruct any other representation of a polytope since the polytope is the convex hull of those vertices. However, this measure is not sufficient in dimension ≥ 4 since the number of edges of a polytope of $\mathbf{n}$ vertices may be $\Theta(\mathbf{n}^2)$.

Another measure of size that at first seems reasonable is the total number of faces, which for a d-polytope with v vertices can be $\Omega(v^{\lfloor d/2 \rfloor})$ (Grünbaum, 1967). However, this size is not necessarily sufficient even to write down all of the faces of the polytope in dimension ≥ 4. In order to write down a k-face, one must represent the boundaries of the face in some manner. Even pointers to the bounding $(k-1)$-faces require too much storage, since for d-polytopes, $d \geq 4$, the total number of incidences of k-faces with $(k-1)$-faces can be $\Omega(\mathbf{n}^2)$ where $\mathbf{n}$ is the number of $(k-1)$-faces of the d-polytope (Grünbaum, 1967).

A measure of size that is more reasonable than the previous two is the total number of incidence relations of the form [*d-face*, *(d–1)-face*, . . . , *edge*, *vertex*] of the polytope, where *d-face*, *(d–1)-face*, etc. are names of specific faces of the polytope. For example, the size of a triangle is six under this measure, since there are six [triangle-edge-vertex] incidence relationships. The size of a polyhedron is the total number of [polyhedron, face, edge, vertex] incidence relationships. This measure of size is reasonable because with this amount of storage one can list the faces of a polytope together with their boundaries by representing each k-face as a list of pointers to bounding $(k-1)$-faces. In addition, this representation enables us to recover incidence relations, which we must be able to do for the algorithms below. We will denote the number of such incidence relations of a polytope $\mathbf{P}$ by $I(\mathbf{P})$.

A consequence of defining the size of a polytope to be the number of incidences is that

$$I(\mathbf{P}) = \sum I(\text{facets of P})$$

That is, the number of incidences of a polytope is equal to the sum of the number of incidences of its facets. This is true because the incidence relations of the polytope are the incidence relations of the facets with the name of the whole polytope prepended. It is not true that the number of faces of a polytope is equal to the sum of the number of faces of its facets, since in the latter case faces get counted repeatedly when they are shared by more than one facet.

With the number of incidences of a polytope as a measure of size, we can construct algorithms that are recursive in dimension and take linear time in the size of the polytope. Suppose an algorithm is defined recursively to work on a polytope by working on each facet of the polytope. If the sum of the sizes of the facets of the polytope is the same as the size of the polytope, then such an algorithm works in linear time whenever the time for one level of the recursion is constant. However, if the sum of the sizes of the facets is larger than the size of the polyhedron, such an algorithm cannot work in linear time. Using the number of incidences, $I(\mathbf{P})$, of a polytope as a measure of size, we will now show how to find the separation of two polytopes in $O(I(\mathbf{P}) \cdot I(\mathbf{Q}))$ time for polytopes of size $I(\mathbf{P})$ and $I(\mathbf{Q})$.

3.2. The Separation Algorithm

In order to find the separation of two polytopes in E^d it is sufficient to find the minimum separation of pairs of faces. However, as we show below, we do not need to find the separation of every pair of faces—we can restrict our search to a subset of the pairs and still guarantee that we find the separation of the polytopes. In particular, we claim that we only need to consider the separation of pairs of faces that are not partly parallel, that is, not linearly dependent. If the faces are not partly parallel, then they intersect in a point or have a mutual normal line, and we claim that we need only consider faces where this intersection point or the feet of the normal lie in the faces. Also, we need only test pairs of faces such that the sum of the dimensions $d_1 + d_2 \leq d$, since otherwise the faces must be partly parallel. We prove these claims below.

The algorithm for finding the separation of polytopes $\mathbf{P}$ and $\mathbf{Q}$ in E^d is the following:

- For each pair of faces **f** and **g** of dimension d_1 and d_2 where $d_1 + d_2 \leq d$ and Aff(**f**)=**F** and Aff(**g**)=**G** do:
  - Determine if the faces are partly parallel, that is, whether **F** and **G** are linearly dependent. If so, skip to the next pair of faces.
  - If $d_1 + d_2 = d$ then **F** and **G** intersect in a single point **p**. Find **p** and determine if $\mathbf{p} \in \mathbf{f}$ and $\mathbf{p} \in \mathbf{g}$ using the point-polytope algorithm. If so, the polytopes intersect.
  - Find the unique mutual normal to **F** and **G** in the space that they span. Find the feet of this normal and determine whether they are in **f** and **g** respectively (using the point-polytope algorithm). If so, the distance between the feet is a candidate for the separation. Remember this distance if it is smallest so far.
- If the separation is non-zero, determine if one polytope is inside the other by using the point-polytope algorithm on a point of each.

Algorithm 2. Finding the separation of polytopes.

Determining whether two moving polyhedra collide is done by constructing the 4-prisms corresponding to the motions and using the polytope separation algorithm above to determine whether the prisms have a non-empty intersection. In order to determine whether any of several moving polyhedra collide, we test them pairwise for collision.

To determine the time and location of collision, we claim that it is sufficient to check every pair of faces that intersect in a single point and find the first one, i.e. the point of intersection with the lowest time-coordinate. We prove this with Lemma 2, below. The only necessary modification to the algorithm above is to keep track of time and location of every one-point intersection of a pair of faces.

Lemma 2. If two moving polyhedra collide, then at the moment of their collision a k-face of each, $k \leq 2$, will intersect in a single point.

Proof. Suppose the polyhedra collide in such a way that at the moment they hit, they intersect in a line or a polygon. Then a vertex of this line or polygon is the point of intersection of a face of each polyhedron. ■

In order to prove that the polytope separation algorithm works we must prove our claim that it is sufficient to find the separation of a subset of the pairs of faces. We do so with a series of lemmas. For the following lemmas assume that **P** and **Q**

are polytopes in E^d and **s** is a line segment connecting them that realizes their separation. Assume also that of all the faces of **P** and **Q** that **s** intersects, **p** and **q** respectively are the faces of lowest dimension. The first lemma is the observation that **s** intersects **p** and **q** in interior points, provided that we allow a vertex to be an interior point of itself.

Lemma 3. **s** intersects **p** and **q** in interior points.

Proof. Suppose **s** intersects **p** or **q** in a boundary point. Then **s** intersects a lower-dimensional face of **P** or **Q**. ■

The next lemma shows that we need only consider faces whose containing flats have a mutual normal, provided that we say a line is normal to a point that it intersects.

Lemma 4. **s** is normal to **p** and **q**.

Proof. Suppose that **s** is not normal to **p** or to **q**. By Lemma 3 we have that **s** intersects **p** and **q** in interior points, and if **s** is not normal to **p** or **q** then **p** or **q** must not be a vertex, so we can slide an endpoint of **s** in some direction and shorten **s**. Therefore **s** does not realize the separation. ■

Thus, we need only consider pairs of faces that have a mutual normal. However, faces that are partly parallel can have many mutual normals; we want to show that it suffices to test faces that have a unique mutual normal. To do so, we must show that if two faces that realize the separation of the polytopes are partly parallel, then the separation is realized by a pair of faces of lower total dimension.

Lemma 5. If two faces are partly parallel, then their separation is realized by a line segment connecting a boundary point of one and a point of the other.

Proof. The separation is realized by a normal line segment, which degenerates to a point of intersection when both faces are translated to the origin. However, since the faces are partly parallel, they intersect in at least a line segment, so a boundary point of one intersects a point of the other. Thus a line segment from a boundary point of one of the untranslated faces to the other realizes the separation. ■

Finally, of the pairs of faces that have a unique common normal, we show that we need only consider pairs for which the feet of the normal are in the faces.

Lemma 6. If Aff(**p**) and Aff(**q**) have a unique common normal but the feet of the normal are not in **p** or **q**, then the separation of **P** and **Q** is realized by a different pair of faces.

Proof. The separation is not realized by interior points of **p** or **q** since if it were, it could not be normal to the points. If it is realized by boundary points, then it is realized by another pair of faces. ■

Thus, in order to find the separation of polytopes we need only consider the separation of pairs of faces that are not partly parallel and such that the point of intersection of the flats or the feet of the mutual normal lie in the faces. Also, we need only test pairs of faces such that the sum of the dimensions $d_1 + d_2 \leq d$. Therefore the algorithm finds the minimum separation of a subset of the pairs of faces that is sufficient to guarantee that the separation found is also the separation of the polytopes.

In the algorithm above we make use of an auxiliary algorithm for determining whether a point is in a d-polytope in E^d, which we call the point–polytope algorithm, and an algorithm for determining whether a line segment intersects a facet in E^d, which we call the segment–facet algorithm. These algorithms are mutually recursive in dimension: the point-polytope algorithm for dimension d makes use of the segment-facet algorithm for dimension d; the segment–facet algorithm for dimension d makes use of the point-polytope algorithm for dimension d–1; and so on. We first present the point–polytope algorithm.

3.3. The Point–Polytope Algorithm

Given a point **p** and a d-polytope **P** in E^d, we now present an algorithm for determining whether the point is in the polytope. The problem is easy in the case d=1. In this case the problem is to determine whether a point **p** lies in some closed interval of the real line [**ab**].

For higher-dimensional cases, we use a straight-forward generalization from E^2 and E^3 to E^d of a well-known algorithm based on the Jordan Curve Theorem. The algorithm is to consider a ray **r** from the point in any direction and count the number of times that the ray intersects the boundary of **P** using the segment-facet algorithm (below) on **r** and each facet of **P**. Since the polytope is bounded and each intersection with the boundary of **P** means entering or leaving **P**, the number of intersections is odd if and only if $\mathbf{p} \in \mathbf{P}$.

In general **r** will intersect the interior of the facets that it intersects. However, if it intersects the boundary of facets then there are special cases to be handled. If **r** intersects the boundary of a facet at a point, then we can find the

lowest-dimensional face **f** that it intersects at that point by following the links from the facet to the incident faces. Once we have found **f**, we can check the sense of the intersection of **r** with each facet incident upon **f**, since we know the inside and outside of a facet. If the sense of the intersection is the same for every facet incident upon **f**, then we count one intersection with the boundary of **P**. If the sense of the intersection is not the same for every facet incident upon **f**, then we count two intersections: the ray enters and exits **P** at that point (or touches **P** but does not enter or exit it). We mark the faces incident upon **f** as "done" so that we do not count the intersection twice.

If **r** lies in the hyperplane spanned by some facet, we use the segment-facet algorithm recursively to count the number of intersections of the segment with that facet. At the boundaries of the facet, we must determine whether the ray is entering/exiting the polytope or just the boundary. That is, we do not count intersections where the ray passes from the interior into the boundary. We count only cases where the ray enters or exits the polytope.

An alternative to handling all of the special cases is picking a random direction and counting the number of intersections of the ray in that direction with the interior of facets. If the ray intersects the boundary of any facet, pick a new random direction and repeat. This takes a small constant expected number of "stabs" for any "reasonable" polytope—i.e. any polytope whose facets are not almost completely boundary.

3.4. The Segment-Facet Intersection Algorithm

Given a line segment **s** and a facet **f** (that is, a $(d-1)$-polytope) in E^d, we now present an algorithm for determining whether the segment intersects the facet. If $d = 1$, the problem is to determine whether a given line segment contains a point on the line. To do this we check whether the point is between the endpoints of the line segment.

In higher dimensions, we first test whether the line segment lies in Aff(**f**) or is parallel to it. If it is parallel, **s** and **f** do not intersect; if the line segment lies in Aff(**f**), then we test **s** with the boundary faces of **f** using the segment-facet algorithm recursively. If **s** and **f** do not intersect we must determine whether **s** is completely inside **f** using the point-polytope algorithm on an endpoint of **s**. Otherwise, the line containing **s** intersects the hyperplane containing **f** in a single point. It remains to check whether the intersection point of **s** and **f** is in **s** and **f**. We do this using the point-polytope algorithm for dimension 1 and $d-1$.

We show with the next lemma that the point-polytope and segment-facet algorithms require linear time in the number of incidences of the polytope, if we consider the dimension of the problem to be a constant.

Lemma 7. The runtime of the point-polytope and segment-facet algorithms for a polytope **P** is O(I(**P**)).

Proof. The algorithms are mutually recursive in dimension. The point-polytope algorithm for dimension d calls the segment-facet algorithm for dimension d or less, and the segment facet algorithm for dimension d calls the point-polytope algorithm for dimension d–1 or less. Therefore the algorithms halt.

In determining whether a point is in a d-polytope or a line segment intersects a d-facet, the recursion occurs only a constant number of times since at each recursive call the dimension is reduced. Therefore in order to determine the runtime of the algorithms we need only consider the runtime of one level of recursion of the algorithms. At the top level of recursion, the point-polytope and segment-facet algorithms both consider each facet only once. Thus, the algorithms take time O(I(**P**)). ■

3.5. Collision Detection

Using the results of the previous sections, we can show that the polytope separation algorithm takes quadratic time in the number of incidences of the two polytopes:

Lemma 8. The polytope separation algorithm for polytopes **P** and **Q** takes O(I(**P**)·I(**Q**)).

Proof. The polytope separation algorithm uses the point-polytope algorithm at most once for every pair of faces of the polytopes. Since the sum of the number of incidences for each face is the number of incidences for the whole polytope, the whole algorithm takes time O(I(**P**)·I(**Q**)). ■

Therefore we can find collisions among several polyhedra in quadratic time:

Theorem. Using the polytope separation algorithm we can find the time and location of the first collision among several polyhedra undergoing linear motion in $O(\mathbf{n}^2)$ time where **n** is the total number of vertices of the polyhedra.

Proof. The polytope separation algorithm requires $O(I(\mathbf{P})\cdot I(\mathbf{Q}))$ time. However, for 4-prisms, $I(\mathbf{P}) = O(\mathbf{n})$ since the bases of the prism are polyhedra and the additional faces are in one-to-one correspondence with the faces of a basis. In fact, the total number of faces of the prism is 3 times the total number of faces of the polyhedron. Thus, finding the time and location of intersection of two polyhedra takes $O(\mathbf{n}^2)$ time where $\mathbf{n}$ is the total number of vertices. Determining whether several polyhedra collide is done by finding collisions of all pairs, which therefore requires $O(\mathbf{n}^2)$ time where $\mathbf{n}$ is the total number of vertices. ■

4. Extensions

The approach of detecting collisions by considering time to be a dimension also works for polyhedra that are rotating or otherwise moving in a non-linear path. However, the volumes of E^4 corresponding to the motions are not polytopes; the vertices trace out curved paths depending on the motion and rotation. Thus there is no notion of skew faces and we cannot find the separation as described above.

However, if two moving polyhedra collide, at the moment of collision two faces still intersect in a single point. This can happen in two ways: by vertex-face contact and by edge-edge contact. Assuming that at the initial time the polyhedra do not intersect, in order to detect collision it is therefore sufficient to find single-point intersections of the 1-manifolds and 3-manifolds swept out by vertices and faces, and of two 2-manifolds swept out by edges.

In the case of edge-edge contact, it is sufficient to find the one-point intersections of the 2-surfaces containing the edges, find the times at which they occur, and test whether the edges intersect at those times. In the case of vertex-face contact, it is sufficient to find one-point intersections of the 1-surface corresponding to the motion of the vertex and the 3-surface corresponding to the motion of the plane containing the face. Then test whether the vertex and the face actually intersect at the time of each intersection point.

5. Conclusion

We have presented an $O(\mathbf{n})$-time algorithm for determining whether two convex polyhedra of total size $\mathbf{n}$ will collide and an $O(\mathbf{n}^2)$-time algorithm for the case of several general polyhedra. Both algorithms use space $O(\mathbf{n})$. The algorithm for the convex case is a generalization of Dobkin and Kirkpatrick's $O(\mathbf{n})$-time convex polyhedron separation algorithm. The algorithm for the general case is a

generalization of the naive separation algorithm for polygons and polyhedra; we generalize it to find the separation of polytopes in E^d. We also show how to find the time and location of intersection in this case.

The algorithm for collision detection in the general case takes only a small constant factor more time than a polyhedron intersection test, since for a polyhedron with **n** vertices, edges, and faces, the prism corresponding to a linear motion has a total of 3**n** faces. Since any collision detection algorithm will have to test for intersections among objects and obstacles at least once, we believe that ours is a practical approach. It is certainly more efficient than any algorithm that repeatedly tests for object/obstacle intersections, and it is not much more difficult to implement than a polyhedron intersection algorithm.

Since the algorithm for collision detection in the convex case uses O(**n**) time, it is clearly order-optimal if we assume that part of the problem is to read in the description of the object. However, it may be possible to improve the collision-detection time if we allow the use of a (possibly large) data structure already in memory. The algorithm for the general case takes the same amount of time as the best known polyhedron-intersection algorithms. Possible extensions include developing the mathematics for objects undergoing nonlinear motions and for extending the method to handle jointed objects such as robot arms. It also may be possible to improve the expected running time for the general case by using a hierarchical representation of the polyhedra.

Acknowledgement

The helpful comments of Charles Dyer are gratefully acknowledged.

References

Boyse, J., "Interference detection among Solids and Surfaces," *Comm. ACM* **22**, 1979, pp. 3-9.

Cameron, S., "A study of the clash detection problem in robotics," *Proc. IEEE Int'l Conf. on Robotics and Automation*, 1985, pp. 488-493.

Canny, J., "Collision detection for moving polyhedra," *IEEE Trans. Pattern Analysis and Machine Intelligence* **8**(2), 1986, pp. 200-209.

Chazelle, B., "Convex partitions of polyhedra," *Siam J. Comput.* **13**(3), 1984, pp. 488-507.

Culley, R. and K. Kempf, "A collision detection algorithm based on velocity and distance bounds," *Proc IEEE Int'l Conf. on Robotics and Automation*, 1986, pp. 1064-1069.

Dobkin, D. P. and D. G. Kirkpatrick, "A linear algorithm for determining the separation of convex polyhedra," *J. Algorithms* **6**, 1983, pp. 381-392.

Ganter, M. A., *Dynamic Collision Detection Using Kinematics and Solid Modelling Techniques*, Ph.D. thesis, University of Wisconsin - Madison, 1985.

Grünbaum, B., *Convex Polytopes*, Wiley, 1967.

Hayward, V., "Fast collision detection scheme by recursive decomposition of a manipulator workspace," *Proc. IEEE Int'l Conf. on Robotics and Automation*, 1986, pp. 1044-1049.

Lozano-Perez, T., "Spatial planning: a configuration space approach," *IEEE Trans. Comp.* **C-32** (2), 1983, pp. 108-120.