Simplifying Parallel and Distributed Simulation with the DUP System

Nathan S. Evans* evans@net.in.tum.de
Chris GauthierDickey† chrisg@cs.du.edu
Christian Grothoff* grothoff@net.in.tum.de
Krista Grothoff* kgrothoff@sec.in.tum.de
Jeff Keene† jefkeene@cs.du.edu
Matthew J. Rutherford† mjr@cs.du.edu

Abstract

This paper presents how the DUP System, a straightforward POSIX-compatible framework that enables programming-language-agnostic parallel and distributed stream processing, can be used to facilitate parallel and distributed simulations. Specifically, we describe two ways of using DUP to utilize available resources for efficient simulation: (1) a straightforward technique for parallelizing multiple runs of an existing simulation program with minimal changes, and (2) *FiDES*, a Discrete-Event Simulation (DES) framework built atop DUP that provides a simple, yet powerful, means of implementing a parallel and/or distributed DES. We then describe a toolset for profiling, debugging and visualization that aids the development of DUP simulations. To support these claims, we present various performance benchmarks that collectively demonstrate how DUP and *FiDES* can make high-performance simulation accessible to everyone.

1. INTRODUCTION

There is a trend in commodity computer hardware toward machines with multiple cores and/or processors, cheap high-performance networking, and inexpensive specialized co-processors. These features, in combination, put a tremendous amount of processing power within the reach of most developers. For scientists, engineers, and business-people working with simulations, access to such computing power promises both larger-scale simulations and faster execution of multiple-trial simulation experiments — but only if software that takes advantage of the available hardware resources can be developed. Unfortunately, parallel and distributed software development remains a challenging task; time devoted to mastering this takes away from the main goal of modeling and simulating phenomena of interest. Furthermore, existing DES frameworks that offer support for parallel and distributed execution require the programmer to develop their simulations using toolkit-specific programming languages and APIs.

In this paper, we present the DUP System[1], a straightforward POSIX-compatible framework that enables programming-language-agnostic parallel and distributed stream processing. The DUP System enables developers to compose applications from stages written in almost any programming language and to run distributed streaming applications across all POSIX-compatible platforms. The DUP System includes a range of simple stages that serve as general-purpose building blocks for larger applications. The DUP System is a general-purpose framework, and we have successfully used it to rapidly parallelize and distribute a wide variety of programs.

In this paper, we focus on uses of DUP that are germane to simulation. In particular, we describe two ways in which we use DUP to utilize available processing resources for efficient simulation. In the first scenario, we describe a straightforward technique to solve the problem of parallelizing multiple runs of a simulation and gathering the resulting output data. Many readers will be familiar with this problem and have undoubtedly spent time developing scripts and other execution harnesses to mitigate this issue. We believe the approach we provide is elegant, powerful, and can be accomplished with minimal changes to existing simulation programs.

In the second scenario, we consider the case where an individual simulation run either takes too long to complete or is limited by the memory available on any one machine. In this case, the developer would like to parallelize the simulation (e.g., to take advantage of multiple cores/processors within a single machine) and/or distribute the simulation (e.g., to take advantage of memory resources available on other machines). To tackle this problem, we have developed *FiDES*, a Discrete-Event Simulation (DES) framework built atop DUP that provides a simple, yet powerful, means of implementing a parallel and/or distributed DES. *FiDES* allows the simulation developer to build their DES out of a flexible number of operating system processes that can be easily moved between machines. The implementation of DES processes within *FiDES* is quite straightforward and can be done in any programming

*Fakultät für Informatik, Technische Universität München
†Department of Computer Science, University of Denver

[1]Available at http://dupsystem.org/

language that supports reading and writing from standard input and output.[2] By providing flexible parallelization and distribution, *FiDES* allows the developer to decide how best to organize their software such that available resources may be utilized. To support the developer in this endeavor, we have developed a suite of decision-support tools to assist with profiling and visualization of DUP program behavior. These can be applied easily to *FiDES* simulations in order to determine how to organize and deploy the parts of the system.

The remainder of this paper is organized as follows: Section 2. provides a high-level introduction to the technical aspects of the DUP system. Sections 3. and 4. present the aforementioned uses of DUP, while Section 5. describes the supporting toolset. Section 6. presents related work, and Section 7. concludes.

2. THE DUP SYSTEM

DUP is a stream processing language designed to allow a broad range of users, from non-programmers to programming experts, access to the benefits of parallel and distributed architectures. The key idea behind the DUP System is the multi-stream pipeline programming paradigm [4]. Multi-stream pipelines are a generalization of UNIX pipelines. Multi-stream pipelines in DUP are composed of **stages**, which are processes that read from any number of input streams and write to any number of output streams. By generalizing UNIX pipelines to multi-stream pipelines, we eliminate the main restriction of the UNIX pipeline paradigm, namely the inherently linear data flow.

A DUP program is organized as a data-flow graph where the nodes of the graph represent operating system processes, and the edges represent directed data-flows between them. The runtime system seamlessly handles the communication setup for pipelines that connect processes — both on the same machine, and across the network. The DUP System also includes useful stages for handling multiple data streams so that, in most cases, the application developer must only write simple deterministic single-threaded programs, without the need for networking or complicated stream handling logic. Stages do not share memory, so there is no possibility for memory consistency issues or data races, and when the data-flow graph is acyclic, there is no possibility for deadlock. As a result, the DUP System allows task- and data-level parallelism to be exploited with a minimum of programmer effort.

The DUP System runs on any POSIX-compatible system and uses both pipes and TCP streams for the communication between stages. Figure 1 illustrates how the components of the system work together. The `dup` command interprets programs written in DUP Assembly language, which specifies

[2] *FiDES* includes a C++-specific library for deserialization (parsing), demultiplexing and serialization of event messages; this minimal language-specific support code could be ported within hours to new languages — if needed for developer convenience.

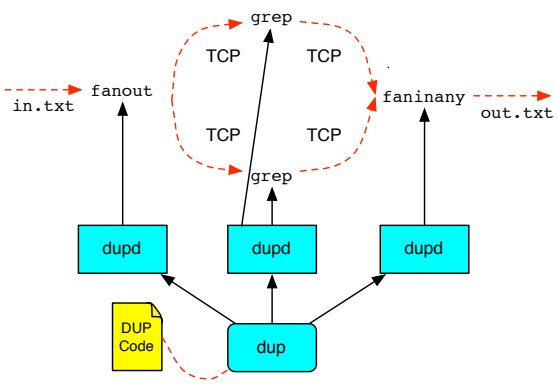

Figure 1. Overview of the DUP System. Red (curved, dashed) lines show data flow. Black (straight, solid) lines correspond to actions by DUP.

precisely how the various stages for the application should be connected. `dup` then communicates this information to `dupd` daemons running on each host. The `dupd` daemons are given information about what stages to execute and how to connect these stages to others. Stages communicate with each other using standard file I/O operations (for example, `read` and `write`), which are available for almost all programming languages.

The main advantages of this approach are that stages can be written in almost any programming language and can communicate with other stages running on different hosts or cores without knowing how they are connected in the global topology. This makes it easy to combine many small components to form large applications. In particular, existing UNIX filters (such as `grep`) can be used within the DUP System without changes. The DUP System also includes stages for common tasks. For example, `fanout` reads data from standard input and writes it to all open and writable file descriptors (except standard error). Many of these generic stages are inspired by and named after similar stages available in CMS [4].

3. PARALLELIZING SIMULATION RUNS

This section describes how DUP can be used to parallelize simple simulations and distribute them across a cluster of machines. In this section, we use a running example of a Blackjack simulation developed independently to evaluate various aspects of the game. The purpose of this example is to illustrate the best-case scenario for the use of DUP for simulation and to provide a first motivation for the system deployment tools presented in Section 5..

3.1. The Blackjack Simulation

The Blackjack simulation (`BlackJackSim`) is a simple Java-based application which, given a set of parameters, simulates Blackjack hands using different player strategies. The

purpose of the simulation is to discover which strategies yield the best results for the player over time. Some of the important parameters to the simulation are the number of hands, the players starting dollar amount, the casino rules and a player strategy. The simulator reads a single line from a file or standard input containing the required simulation parameters and then runs the simulation, writing the results to standard output. The simulator then reads another set of parameters from standard input and simulates those. This continues until there are no more inputs provided to standard input. This design allows the simulation to be run naturally within the DUP system, as it operates in a stream-oriented fashion.

3.2. Parallelizing with DUP

The easiest way to parallelize the Blackjack simulation is to dynamically split the single input file by newline delimiters using DUP's `deal` stage (which distributes the lines from one input stream round-robin to *n* output streams) and then pass these sets into multiple `BlackJackSim` processes. The resulting output lines from the `BlackJackSim` processes can then be joined into one meta simulation file using the `faninany` stage (which merges lines from *n* input streams into one output stream). Input and output are delineated by newlines, and both `deal` and `faninany` operate on lines by default. The graphical representation of the resulting DUP configuration is shown in Figure 2.

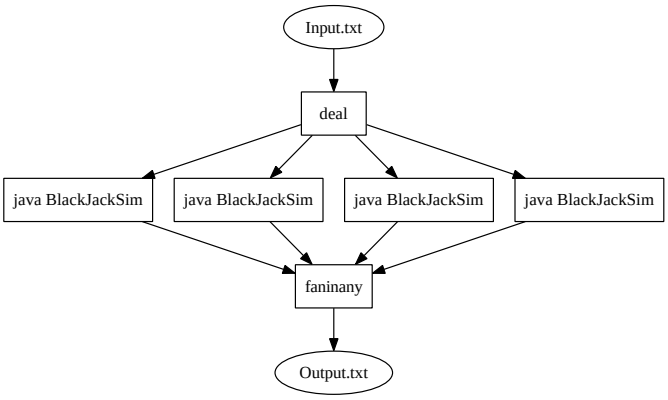

Figure 2. Illustration of a parallelized Blackjack simulation setup. The simulation parameters are read from `Input.txt` and sent to four `BlackJackSim` processes. The output from these four processes is combined and written to `Output.txt`.

The DUP Assembly code used to implement the parallel simulation shown in Figure 2 is virtually the same for parallel execution on one host and distributed execution using multiple hosts. The code only differs in the hostnames specified for the various `BlackJackSim` stages. However, a user running this type of simulation with DUP need not worry about the low level DUP specification; they simply need to fill in an XML file and execute the provided driver to generate the DUP Assembly code. Listing 1 shows the high-level skeleton XML specification needed for running a distributed version of the `BlackJackSim` simulation.

Listing 1. Example XML Specification for running a distributed simulation of this type.

```
<dup_simulation>
  <local_data_dir >/path </local_data_dir>
  <remote_data_dir >/tmp</remote_data_dir>
  <simulation_command>java  BlackJackSim
  </simulation_command>
  <input_file >Input.txt </input_file>
  <ssh_username>user </ssh_username>
  <simsOut2Stdout >1</simsOut2Stdout>
  <localOutFile >out.dat </localOutFile>
  <remoteDUPPath >/path </remoteDUPPath>
  <start_dupds >1</start_dupds>
  <control_host >host0 </control_host>
  <hosts>
    <host>
      <hostname>host1 </hostname>
      <port >55555</port>
    </host>
    ...
  </hosts>
</dup_simulation>
```

3.3. Optimization

The basic DUP runtime system leaves the developer with the task of selecting a mapping of stages to hosts. This choice is not always obvious, especially for more complex simulations, since resource constraints such as CPU utilization, memory consumption and network bandwidth all need to be considered in order to maximize performance.

The Blackjack simulation is primarily limited by the CPU. Using more `BlackJackSim` stages only gives better performance as long as there are more processors or cores idle on the host. However, even in this simple case, developers might want to confirm that performance is not limited by network bandwidth. Furthermore, in the age of multi-core machines and processor features like hyper-threading [6], it is not always obvious what the optimal number of stages per host is.

For the simple Blackjack simulation, it is relatively easy to find the optimal allocation of processes to machines by trying out combinations manually. Figure 3 shows that using the right allocation strategy, the Blackjack simulation scales linearly with the number of available hosts. Naturally, this cannot be expected to be the case in general, especially for more demanding simulations. Section 5. describes a convenient and systematic approach for resource allocation within the DUP

system that makes it easy to perform resource allocation and optimizations for all kinds of DUP applications.

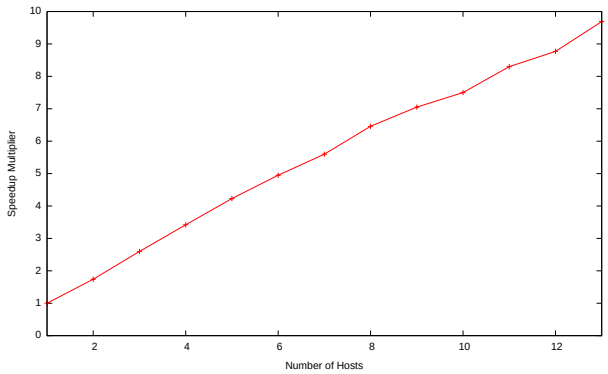

Figure 3. This figure shows the speedup achieved when running our Blackjack simulation on varying numbers of machines (two cores per machine).

4. *FIDES*

The approach presented in the previous section is appropriate for situations in which individual simulation runs can be handled by a single computer. However, part of the promise of access to powerful parallel and distributed hardware is to enable "larger" simulations than have previously been possible — this means parallelizing and distributing single simulation runs. Parallel and distributed simulation engines are not new; however, their complexity is a barrier to entry for non-specialists. The goal of *FiDES*, therefore, is to provide a parallel and distributed Discrete-Event Simulation (DES) framework that can be easily mastered and is straightforward to tune and adapt to different hardware platforms.

As with all DES engines, the principle abstractions provided to the developer are *process* and *event*. A central goal of *FiDES* is to provide an architecture that does not require *a priori* knowledge of how a particular simulation will be decomposed to support parallelism and distribution. To achieve this goal, a simulation written for *FiDES* can be executed entirely within a single operating system process (of course, this cannot be parallelized or distributed) or it can be executed, using DUP, as multiple operating system processes with one or more simulation processes per program. When executed on a single system, the operating system will schedule the *FiDES* processes to execute on different cores/processors (thereby leveraging more CPU power). If memory consumption becomes a problem, the configuration can be tweaked to schedule the processes for execution on different machines. We use the term *FiDES stage* to refer to an operating system process, and *simulation process* to refer to the DES processes running inside of *FiDES* stages.

For developer convenience, *FiDES* includes a simple C++ API supporting the basic concepts of process and event and two different simulation engines: monolithic and parallel. This support library is implemented in less than 450 lines of actual C++ code and primarily supports the deserialization (parsing), demultiplexing and serializing of events that is required for the parallel simulation. Porting this library to other programming languages could be accomplished in a matter of hours.

4.1. Events

Events consist of a type attribute and a collection of name/-value pairs representing other application-specific values associated with the event. For language-independence, names and values are simply strings as far as *FiDES* is concerned. Of course, developers are free to provide application-specific wrappers to support stronger typing and ease of use in their applications. The C++ API represents events using the (trivial) `Event` class.

4.2. Simulation logic

The simulation logic is expected to process a stream of incoming events, including "advance-time" events, and generate the respective state changes and output events. To facilitate routing of event messages, *FiDES* requires each simulation process to have a unique identifier that is also specified in events destined for this process.

The *FiDES* C++ API provides a `Process` class that provides high-level abstractions for writing the simulation logic. Derived classes must implement the `processEvent(const Event&)` method that is called by the framework when a scheduled event is due to be processed. Derived classes can optionally override the `init()` method, which is called exactly once by the framework before any events are processed — this is typically used to schedule initial events for the simulation or perform other initialization.

Each `Process` instance maintains its own virtual clock, updated in response to an "advance-time" message from the simulation engine or when the simulation process itself advances time to represent processing or busy time while handling events. Remote and local (self) events are scheduled using one of several convenience variants of the `scheduleEvent()` method. These methods perform differently depending on the simulation engine being used.

Once the application-specific `Process` classes are defined, they can be instantiated and bound together into a single monolithic *FiDES* stage using the `MonolithicEngine` class. This engine is simple to use: (1) an instance of the `MonolithicEngine` class is created; (2) simulation processes are instantiated and added to the engine instance; and (3) the simulation is executed. The monolithic engine is included to facilitate development

of the application-specific elements of a simulation (i.e., for testing) and to provide simulation developers with a fall-back position in the event that parallel/distributed simulation is not warranted for some simulation scenarios.

4.3. Parallel Engine

Parallel engine stages are the entities in a distributed *FiDES* simulation that run the application-specific simulation process logic. Each parallel *FiDES* stage can be responsible for any number of simulation process instances. A parallel engine must parse event messages, execute the respective processing code and serialize event messages for other *FiDES* stages. The `ParallelEngine` class provides a sample implementation of this simple core logic for C++. It is typically used in conjunction with a `main` function that processes a configuration file specifying which simulation processes should be run by the *FiDES* stages.

4.4. *FiDES* Runtime Architecture

Figure 4 depicts a simplified logical organization of a *FiDES* parallel simulation. In this figure, the grayed boxes represent elements of the FiDES/DUP framework, while the white boxes represent *FiDES* stages containing simulation processes; lines with arrow-heads represent data flows. Conceptually, the `fanin` connected to the `fanout` implements a message bus between all *FiDES* stages — in practice, the implementation is more sophisticated than this, employing different strategies to filter and otherwise reduce unnecessary data exchange. The white boxes labeled "PE 1", "PE 2", etc. represent programs executing the `ParallelEngine` simulation engine.

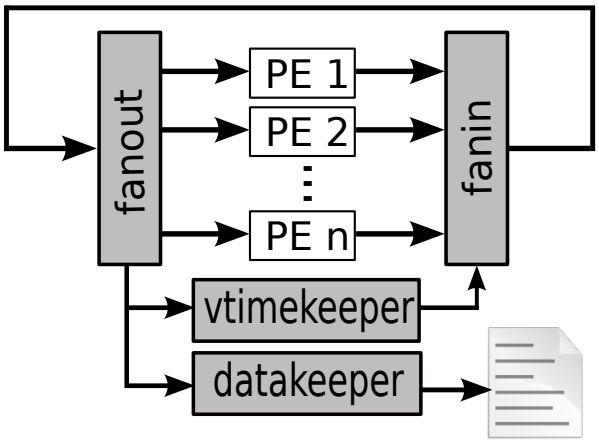

Figure 4. Conceptual organization of a *FiDES* parallel execution. Grayed boxes represent elements of the *FiDES* infrastructure; white boxes represent application-specific processes.

FiDES uses a conservative synchronization strategy with a central virtual timekeeper process. While this limits exploitable parallelism, avoiding the need to support roll-backs simplifies the development and integration of simulation components written in many languages by keeping the amount of language-specific support code minimal.

The simulation starts when the `vtimekeeper` simulation process sends a `VTIME 0` message (delimited by a newline) to all the *FiDES* stages. The stages deserialize this message and update the virtual time of all the simulation processes it contains (initializing as necessary). During initialization, some simulation processes may schedule events on other simulation processes — the events are simply serialized by the *FiDES* stage and passed off to the message bus, where they are handled by the *FiDES* stage that contains the designated destination process. The `vtimekeeper` is also notified of the time at which an event is scheduled to occur so it can send a `VTIME` message appropriately.

The `vtimekeeper` process advances virtual time when all simulation processes have finished processing the previous `VTIME` message. Simulation processes indicate this by sending a `VTACK` message.

At any time, simulation processes can send a `STATE` message with a snapshot of relevant elements of their internal state. These messages are handled by the `datakeeper` processes and typically saved off to files for later analysis.

Message	Source	Destination
`VTIME`	`vtimekeeper`	`ParallelEngine`
`VTACK`	`ParallelEngine`	`vtimekeeper`
`EVENT`	`ParallelEngine`	`ParallelEngine`
`STATE`	`ParallelEngine`	`datakeeper`

Table 1. Summary of *FiDES* messages used to communicate between the processes in a parallel simulation execution.

Table 1 summarizes the major communication messages exchanged between elements of a *FiDES* parallel simulation. These messages are serialized into a newline-delimited stream of text messages with a simple, easy-to-parse format.

Obviously, the flexible design of the parallel execution allows for many configuration options: the developer has to decide on the number *FiDES* stages and which type(s) to group together within each. The system processes shown in Figure 4 can be located on different machines (in a distributed scenario). As all of these decisions can impact the runtime performance of the simulation, we have developed a suite of tools for analyzing performance, as described in the next section.

5. DUP OPTIMIZATION AND ANALYSIS

Distributed and parallel systems are, in general, difficult to optimize, and simulation designers without a strong computer

science background can easily be overwhelmed when trying to get simulations to run faster. To make this process easier, we have developed profiling and analysis tools which help determine the performance and interactions of individual system components. We assume an iterative approach, where a user creates a basic distributed pipeline in DUP and then uses the analysis tools to discover the characteristics of each stage and its interactions. This data can then be used to schedule stages for better performance.

Our performance analysis tools treat stages as black boxes; they can only observe the behavior of each stage from the point of the operating system. They monitor memory and CPU usage, network bandwidth and data, and system calls (using `strace`). Additionally, I/O behavior is observed by inserting stages into the data flow graph that monitor data flow between the original stages. The various instrumentations are run independently over multiple profiling runs to minimize interference. We currently do not have any sophisticated support for alignment of traces, other than system times.

The DUP profiling infrastructure also creates standalone configurations that are run on data captured by the full system run. This allows profiling of individual components without interference from other system components. This allows a system designer to see best case performance of individual stages and also provides a bound on the theoretical performance of the overall system (by adding the individual stage runtimes together). We provide the ability to record all data transmitted between stages (to be played back offline or directly analyzed). We can also record standard runtime metrics such as CPU usage, memory consumption, system calls, and timing information both on- and offline. Since one of the bottlenecks that may be encountered is network bandwidth and latency, we also record the throughput needed for each stage in the system as well as bandwidth between possible hosts. Since collecting so much data can be overkill, mixing and matching of what is gathered is also supported.

To make the collected data more useful to a system designer, the profiling tools can graph this data and merge the plots into a single web page.

We will now present two case studies where we use the profiling infrastructure to analyze and improve performance of the Blackjack and FiDES simulations described in the previous sections.

5.1. Case Study - Blackjack

In order to determine performance bottlenecks of a simulation, we first need to discover the CPU and memory characteristics of each stage. Each stage is first examined in isolation, where interactions between stages do not exist. This gives baseline characteristics of the stages, which can be used to decide what the bottlenecks are for the system. For instance, our blackjack simulations require very little memory (see Table 2). Table 2 further indicates that the stages used in the simulation for sending simulation parameters and merging results (the `deal` and `faninany` stages) require so little CPU time that they are unlikely to be bottlenecks in our simulation. This indicates that memory is not a limiting factor, but CPU usage is for these simulations.

Process	Mem (MB)	% Time
`deal`	0.33 MB	<0.001%
`worker1`	49.96 MB	19.999%
`worker2`	49.66 MB	19.875%
`worker3`	65.15 MB	19.877%
`worker4`	52.24 MB	19.751%
`worker5`	50.33 MB	20.354%
`faninany`	0.39 MB	<0.001%

Table 2. Percentage of system memory (max) used by each process during its lifetime, percentage of total CPU time for each process compared to total simulation, and percentage of CPU used when run in isolation.

Because the Blackjack simulation profiling data indicate that the simulations are CPU-bounded stages, performance should be improved simply by distributing the worker processes to separate hosts. Concrete speedup results are shown in Figure 3 and the reasons for those results are explained below. In this example, we fix the number of worker stages at five, vary the number of hosts used, and plot real time CPU usage when these stages are distributed across one, two and five machines. Real time CPU usage means that interactions between stages can be seen.

Figure 5 shows the CPU usage when all worker stages are run on a single host, two hosts and five hosts. The worker stages when scheduled on a single host must share the available CPU and each get around 40 percent of the CPU on average as opposed to 100 percent when run in isolation. Figure 5 also shows the same five worker stages scheduled on two hosts (three at one, two at the other). This shows that `worker2` and `worker4` (scheduled together on a dual core machine) each get roughly 100 percent of each core, finishing faster than the three workers scheduled on the other host. Figure 5 finally shows the extreme case where each worker is scheduled on a separate host. Here we get the best performance of the three, as each worker can use all of a single core's resources. Of course, in this final case the second core on each host is not utilized at all which means the most efficient use of the available resources is likely to schedule two workers (or more) at each available host.

While improving the layout of this particular simulation is a bit trivial because we knew that it was a CPU-bounded application, the same approach can be used to discover the behavior of stages that are not as easy to predict. Our profiling tools provide memory and network usage data and graphs

(left out for brevity) so that the interactions of more complex systems can also be discovered. This allows the designer of the simulation to decide how best to utilize available resources.

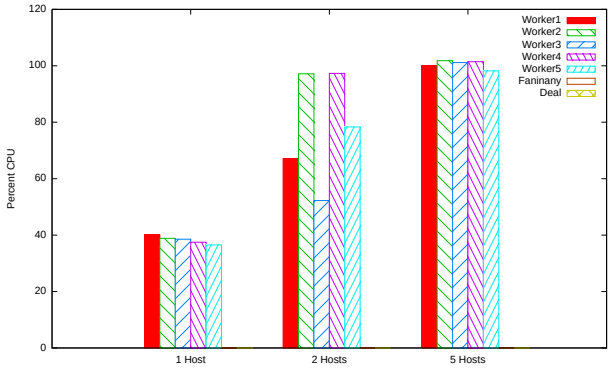

Figure 5. Average CPU usage of Blackjack simulation running on 1, 2 and 5 machines

5.2. Case Study - FiDES

A logistic queuing simulation very similar to SSIM [1] was used as an example *FiDES* application. This simulation uses three types of *FiDES* stages: **manufacturers** (goods producers), **retailers** (get bulk goods from manufacturer), and **consumers** (buy goods from retailers). This simulation is more complex, and improving performance is not as straightforward as with the Blackjack simulation. *FiDES* allows us to split up the manufacturer, retailer and consumer simulation processes arbitrarily among *FiDES* stages.

The first simulation is run on a DUP configuration with one *FiDES* stage for each type of simulation process. Table 3 shows that, as with the Blackjack simulations, memory is not an issue — the CPU is the bottleneck when run in this configuration. In the next step, we try to speed up the simulation time by increasing the number of *FiDES* stages and scheduling multiple stages for each host available. But, as shown in Figure 6, while increasing the number of FiDES stages and distributing the workload initially decreases total runtime, as more FiDES stages are added, the performance gain decreases.

Using the bandwidth profiling graphs and CPU usage, we determined that the cause is the "global bus" that *FiDES* employs. Any data sent from any virtual process is broadcast to every other virtual process. Essentially, the output from any single stage is sent to every other stage. This means that increasing the number of stages also increases the amount of data sent over the network. Since sending data over network links is slower than data sent via pipes, this increases latency and adversely effects our performance. Once we increase the number of FiDES processes per machine beyond one, the cost of sending and processing the data causes the time for the simulation to run to begin to increase, as FiDES processes are waiting for the signal to run instead of running.

Process	Mem (MB)	% Time
`fanin`	0.41 MB	1.98%
`fanout`	0.41 MB	7.86%
`timekeeper`	0.74 MB	2.45%
`manufacturer`	0.98 MB	26.74%
`retailer`	41.89 MB	26.41%
`consumer`	30.74 MB	34.55%

Table 3. Percentage of system memory (max) used by each process during its lifetime and percentage of total CPU time for each process compared to total simulation time. These data are for a 6,300 virtual process simulation with 3 FiDES processes (one per virtual process type).

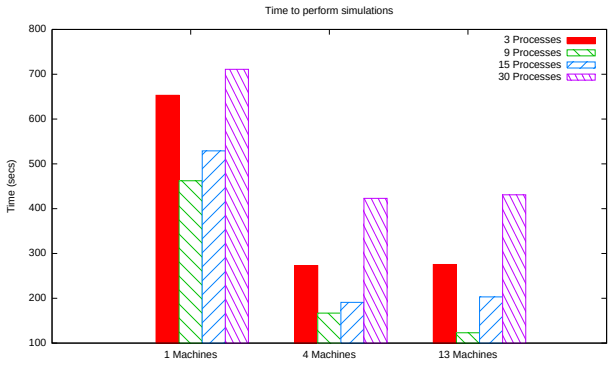

Figure 6. Time to run a 6,300 virtual process simulation varying the number of FiDES processes and the number of machines.

6. RELATED WORK

The grand vision for component-based simulation frameworks is to provide component libraries that can be easily customized and composed to quickly build custom simulations [8]. Virtually all existing simulation frameworks attempt to enable composition by providing either a new language [11, 12] or an API [3, 5, 8, 9, 10] (which is often combined with a GUI for composition) that components must utilize. The core of the simulation framework then provides execution (in particular scheduling), coupling and analysis capabilities for the simulator.

In contrast, *FiDES* is much closer to the DEVS formalism [13]. An atomic DEVS model is a tuple of input events, output events, states, state changes due to progress in time, internal events, and transition functions describing under which conditions output events are generated. The DEVS formalism

combines atomic DEVS into coupled DEVS by specifying how input and output events are coupled.

In *FiDES*, atomic DEVS are essentially represented as processes. This has numerous advantages. Using separate processes liberates components from their particular simulation API or simulation language and enables the composition of components written in many different languages. At this point, many different simulation frameworks exist that largely differ in the host language (for example, CoSMoS [10] uses Java, and DEVS# [3] uses C#); some frameworks [7, 11] even provide APIs for multiple languages (for example, Mimosa [7] supports Java, Scheme, Python and Smalltalk). However, the approach of using language-based APIs inherently limits which simulation components can be combined.

In addition to enabling composition of components written in any language, DUP and *FiDES* also facilitate hierarchical composition particularly nicely. Hierarchical composition is also facilitated since an atomic DEVS "process" can be anything from a simple state machine to another *FiDES* simulation. This way, *FiDES* can even be combined with existing simulation systems that provide an API that is compatible with the atomic DEVS model.

Some simulation frameworks include support for parallel and distributed execution [2, 9]. For example, the SIMA simulation environment [9] offers parallel (but not distributed) execution of discrete event simulations. *FiDES* is also unique in that it enables parallel and distributed simulation with a minimal and portable run-time system; the DUP runtime is only a few thousand lines of C++ code and it runs on any POSIX system.

As far as performance is concerned, the importance of partitioning and mapping simulation tasks in distributed environments is well known [2]. However, to the best of our knowledge, *FiDES* is the first discrete event simulation framework with built-in profiling and resource allocation capabilities.

7. CONCLUSION AND FUTURE WORK

We have presented the DUP System, a language-agnostic system which allows users to easily create correct parallel and distributed programs. The main purpose of the DUP System is to promote productivity in programming and system design. Simulation designers can leverage the DUP System in two ways: running many small simulations across multiple hosts or distributing large simulations that cannot be run on a single system. The DUP System provides simple tools to automatically distribute small simulations and includes *FiDES*, which can be extended to simulate more complex problems. Utilizing DUP for our own simulations has shown real performance increases. The DUP System also includes profiling and debugging tools which can help designers better understand their systems bottlenecks and guide their optimization work. In the future, we intend to extend these profiling and debugging tools to automatically perform certain optimizations. Unlike other parallel and distributed simulation systems, DUP requires no specific language or language extensions to be used in the main simulation code; in fact, DUP can be used in conjunction with legacy code across heterogeneous systems. The DUP System is useful as lightweight middleware for stream-oriented simulations.

Acknowledgements

The authors would like to thank Nathaniel Sandford for allowing us to experiment with his Blackjack simulation, and Min Qi and Craig Ritzdorf for their contributions to early versions of DUP.

REFERENCES

[1] Antonio Carzaniga. SSim — A Simple Discrete-Event Simulation Library. `http://www.inf.usi.ch/carzaniga/ssim/index.html`, 2005.

[2] G. Chiola and A. Ferscha. A distributed discrete event simulation framework for timed petri net models. Technical report, Series of the Austrian Center for Parallel Computation, ACPC/TR, 1993.

[3] Moon Ho Hwang. *Modeling and Simulation using DEVS#*. `http://xsy-csharp.sourceforge.net/DEVSsharp`, first edition, May 2007.

[4] IBM. *CMS Pipelines User's Guide*. IBM Corp., `http://publibz.boulder.ibm.com/epubs/pdf/hcsh1b10.pdf`, version 5 release 2 edition, Dec 2005.

[5] Edward A. Lee. Ptolemy ii. `http://ptolemy.eecs.berkeley.edu/ptolemyII/`.

[6] Deborah T. Marr, Frank Binns, David L. Hill, Glenn Hinton, David A. Koufaty, J. Alan Miller, and Michael Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1): 1, 2002. ISSN 1535864X.

[7] Jean-Pierre Müller. *MIMOSA user's manual*. Cirad, 1.3.0beta edition, Dec 2009.

[8] Herbert Praehofer, Johannes Sametinger, and Alois Stritzinger. Concepts and architecture of a simulation framework based on the javabeans component model. In *Proceedings of WEBSIM99, 1999 International Conference On WebBased Modeling & Simulation*. Elsevier, 1999.

[9] Hassan Rajaei. Sima: an environment for parallel discrete-event simulation. In *Simulation Symposium, 1992. Proceedings. 25th Annual*, pages 147–155, Apr 1992. doi: 10.1109/SIMSYM.1992.227567.

[10] Hessam S. Sarjoughian and Vignesh Elamvazhuthi. *CoSMoS 2.0.0 Guide*. Arizona Center for Integrative Modeling and Simulation, 2009.

[11] Andras Varga. *OMNet++ User Manual*, version 4.0 edition, 2009.

[12] P. Wonnacott and D. Bruce. The apostle simulation language: granularity control and performance data. In *Tenth Workshop on Parallel and Distributed Simulation*, pages 114–123, 1996.

[13] Bernard Zeigler. *Multifacetted Modelling and Discrete Event Simulation*. Academic Press, Boston, 1984.