Application of Parallel Discrete Event Simulation to the Space Surveillance Network

David Jefferson
Lawrence Livermore National Laboratory
James Leek
Lawrence Livermore National Laboratory

ABSTRACT

In this paper we describe how and why we chose *parallel discrete event simulation* (PDES) as the paradigm for modeling the Space Surveillance Network (SSN) in our modeling framework, TESSA (Testbed Environment for Space Situational Awareness). DES is a simulation paradigm appropriate for systems dominated by discontinuous state changes at times that must be calculated dynamically. It is used primarily for complex man-made systems like telecommunications, vehicular traffic, computer networks, economic models etc., although it is also useful for natural systems that are not described by equations, such as particle systems, population dynamics, epidemics, and combat models. It is much less well known than simple time-stepped simulation methods, but has the great advantage of being time scale independent, so that one can freely mix processes that operate at time scales over many orders of magnitude with no runtime performance penalty.

In simulating the SSN we model in some detail: (a) the orbital dynamics of up to 10^5 objects, (b) their reflective properties, (c) the ground- and space-based sensor systems in the SSN, (d) the recognition of orbiting objects and determination of their orbits, (e) the cueing and scheduling of sensor observations, (f) the 3-d structure of satellites, and (g) the generation of collision debris. TESSA is thus a mixed continuous-discrete model. But because many different types of discrete objects are involved with such a wide variation in time scale (milliseconds for collisions, hours for orbital periods) it is suitably described using discrete events.

The PDES paradigm is surprising and unusual. In any instantaneous runtime snapshot some parts my be far ahead in simulation time while others lag behind, yet the required causal relationships are always maintained and synchronized correctly, exactly as if the simulation were executed sequentially. The TESSA simulator is custom-built, conservatively synchronized, and designed to scale to thousands of nodes.

There are many PDES platforms we might have used, but two requirements led us to build our own. First, the parallel components of our SSN simulation are coded and maintained by separate teams, so TESSA is designed to support transparent coupling and interoperation of separately compiled components written in any of six programming languages. Second, conventional PDES simulators are designed so that while the parallel components run concurrently, each of them is internally sequential, whereas for TESSA we needed to support MPI-based parallelism *within* each component.

The TESSA simulator is still a work in progress and currently has some significant limitations. The paper describes those as well.

1. DISCRETE EVENT SIMULATION FOR MODELING THE SSN

The TESSA (Testbed Environment for Space Situational Awareness) Simulator is a scalable, conservative parallel discrete event simulation platform built by LLNL. It is a general purpose simulator, but with some specific capabilities allowing it to run our TESSA simulations of the Space Surveillance Network [1]. It is designed to run on our standard x86 Linux cluster machines, and both the software layers over which it is built, and also the simulation algorithm itself, are known to scale comfortably up to several thousand parallel nodes and to tens of thousands of processes. So far, though, the TESSA simulator has needed to run only a few hundred processes in parallel. In this paper we will describe the TESSA simulator, its scope and limitations, and our future plans for it.

Most scientific simulations are designed to be *time-stepped*, so that the outer loop of the algorithm consists of (1) stepping simulation time forward by a constant delta, (2) updating the state of all parts of the model accordingly, and (3) repeating until completion. Time stepped simulations are very simple to understand and visualize, simple to

implement, and simple to parallelize and synchronize, at least at small scale. And for continuum models time stepped simulation fits naturally with many of the classical numerical algorithms used for solving partial differential equations.

But there are also many limitations and weaknesses of the time stepped simulation paradigm that make it disadvantageous under a variety of circumstances. They arise when a model has an irregular, statically unpredictable interaction pattern in space and time, or exhibits behavior at multiple time- and space-scales, or is composed from many independently developed models that are federated into a single simulation. In this paper we argue that these complications generally apply to modeling of the space surveillance network (SSN) and we suggest adopting an alternative to time-stepped simulation, a paradigm known as *discrete event simulation (DES)*. We will describe DES in general, and then describe the parallel discrete event simulator (PDES) we have built specifically to model the SSN, and explain its unusual features.

Discrete event simulation is not new. It has been around for 50 years, and has been used for many defense applications, as well as simulations of many kinds of discrete system including computer and network systems, vehicular traffic systems, particle systems, queuing systems, population dynamics, and epidemiological studies, to name a few. DES is called for when (a) the system can be decomposed into *discrete interacting objects* (also known as *logical processes*), (b) the system being modeled is dominated by *discontinuous* state changes, and (c) the simulation times of those state changes are *not statically known*, but must be calculated online as the simulation progresses. Generally speaking, systems that cannot be described by equations or other compact mathematical forms are likely candidates for DES. Most artificial systems (as opposed to natural systems) are best modeled that way. But in spite of wide applicability, DES is still not well understood by many people who build simulations.

In Fig. 1 we illustrate the relationship among various simulation paradigms. On the left branch is the entire tree of continuous simulation methods that we do not detail here. The right branch is devoted to discrete simulations in which there is no equational model in the background and no presumption of continuous state change through time, but rather the state is modeled as changing discontinuously. Discrete simulations can be further divided into *time stepped* and *event-driven* (DES) methods. In time-stepped models the fundamental requirement is that the simulation times at which the state changes are globally known before hand, in fact usually statically known, and almost always those times are evenly spaced with a constant time difference between them. In event-driven models, however, the times at which state changes occur are dynamically computed as the simulation progresses, and there is no advanced knowledge of when those times will be and no presumption that they are evenly spaced.

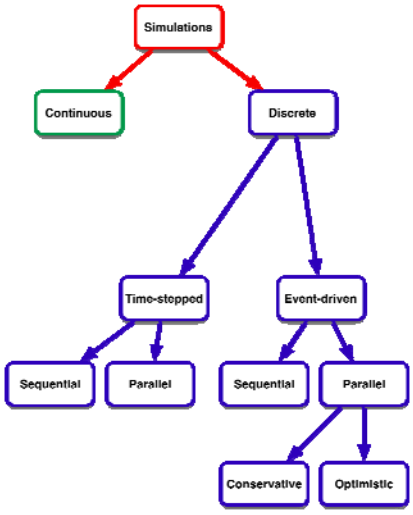

Fig. 1: Classification of simulation paradigms

As Fig. 1 show, both time stepped and discrete event simulations can be done in parallel. But while parallel time-stepped simulation is straightforward and closely resembles its sequential form, parallel DES methods are non-obvious and are very much more complex that the corresponding sequential algorithms. As Fig. 1 also shows, there

are two fundamental families of parallel DES algorithms, *conservative* and *optimistic*, and we will have more to say about them later.

A DES model, whether sequential or parallel, is composed of a flat set of *objects*, where the word "object" is used in the sense of object-oriented programming (with classes, fields, methods, inheritance, etc.), except that for parallel execution the objects should also be thought of as *processes*, with separate address spaces and appropriate overhead costs for creation, destruction, and inter-object communication and synchronization. The *state* of the model as a whole is just the collection of the (nonoverlapping) states of all of the objects. Each of the discontinuous state changes that occur during the simulation is called an *event*, and we further require that each event is a change in the state of just one object. An *event method* is a method $m(t, a_1, \ldots, a_n)$ associated with an object p such that $p.m(t, a_1, \ldots, a_n)$ computes the event at time t, i.e. computes the change in state for object p at simulation time t using arguments $a_1, \ldots, a_n$. But event methods, besides computing the state change in one object typically also *schedule* one or more additional events to be executed at other objects. Thus event $p.m(t, a_1, \ldots, a_n)$ may schedule event $q.n(t', b_1, \ldots, b_r)$ to be executed at object q at a strictly later simulation time $t' > t$. An event $p.m(t, a_1, \ldots, a_n)$ has a *direct causal relationship* with the next later event that occurs in the same object p, and also with any events that it schedules for later simulation times. A sequence of events $e_1, \ldots, e_q$ each having a direct causal relationship with the next forms a *causal chain* by transitivity. Each event in a causal chain is a *causal antecedent* of all of the following ones.

A *sequential* DES is organized at runtime around a central shared priority queue containing all scheduled events, and it is called, appropriately enough, the *event queue*. An event $p.m(t, a_1, \ldots, a_n)$ in the event queue is said to be *scheduled* for simulation time t, and t is the *time stamp* for that event. A sequential simulation starts with one or more initial events in the event queue. Thereafter the simulation proceeds by repeatedly removing the event with the lowest time stamp from the event queue, setting simulation time to be the timestamp of that event, and then executing the event, which usually has the side effect of inserting one or more future events into the event queue. The simulation terminates normally when some threshold simulation time is reached, although it may also terminate abnormally of the event list is empty.

In the case of the TESSA simulation of the SSN, however, we need a *parallel* discrete event simulation (PDES), as we will discuss in the next section. For PDES the objects in the simulation need to be able execute in parallel, and generally without any shared memory. (Special mechanisms can be used when there is shared memory, but the resulting methods are not scalable.) Because there is no shared memory, there can be no central shared event queue, so the event queue must be distributed, with each object maintaining a priority queue of the events scheduled for it. And when one object schedules and event $p.m(t, a_1, \ldots, a_n)$ for another object p, it does so by sending p an *event message* that indicates the simulation time t (the timestamp) at which it should be executed, along with the method name m and parameters $a_1, \ldots, a_n$ as needed. In Fig. 2 we show a generic diagram of a few circular objects in a PDES and the inter-object communication arcs along which they send event messages to one another. Each object maintains its own simulation clock and state, along with a priority queue of event messages sorted by the event's time stamp. The central highlighted object in Fig. 2 has reached simulation time 35.1 and has four queued event messages sorted by time stamps that range from 37.2 to 61.0.

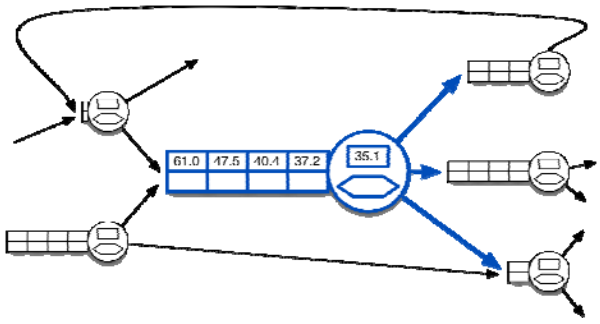

Fig. 2: Objects in a PDES send time stamped event messages enqueued in time stamp order

Each object in a PDES must act on the event messages that are sent to it in strictly increasing time stamp order, even though they do not in general arrive in time stamp order. If all objects do that, then (ignoring ties in simulation time) the computational outcome is identical to what would have happened if the events were executed sequentially in increasing order. Making sure that every object executes its event messages in increasing time stamp order is the central synchronization issue in PDES, to which we will return in section 3 below.

Parallel discrete event simulation is a complex and in some ways counterintuitive field. Usually a naïve first attempt at a parallel DES algorithm starts with the *Synchronous Rule* that all of the objects must stay synchronized in simulation time, so that in any instantaneous global snapshot of the computation all of parallel objects will have simulated forward to the exact same simulation time. This is equivalent to requiring that for all t, each event at simulation time t must complete execution before any other events at times $t' > t$ even start, and thus events will be executed globally in strictly increasing simulation time order.

The Synchronous Rule seems to makes sense by analogy with the way parallel time-stepped simulations work. But as appealing as it may seem, it is the wrong thing to do to achieve good parallel performance in most discrete event simulations because it allows two events to be executed in parallel only when they are scheduled for the *exact same simulation time*. Since in most discrete event simulations event times are dynamically calculated floating point values, often involving random variates, it is rare to have two or more events scheduled for the exact same simulation time, and hence rare that more that one event can be executed in parallel. Under the Synchronous Rule, there is essentially no actual event parallelism possible at all!

For that reason we abandon the Synchronous Rule in all useful PDES algorithms. In general, two events can execute in parallel even if they are far apart in simulation time as long as there is no causal chain from one event to the other (and sometimes even when there is). We can get much better parallel performance if objects are not constrained remain synchronized in simulation time, but in any given snapshot some will be ahead in simulation time and some will lag behind. There is usually no general bound on how far ahead one part may be in simulation time than another. Which objects are ahead and which behind often changes dynamically as well. Different PDES algorithms all agree on this point, but differ on the details of how they schedule events.

2. WHY IS DISCRETE EVENT SIMULATION APPROPRIATE FOR THE SSN?

The Space Surveillance Network is a worldwide network of radar and telescope sensors, computers, databases, and other components involved in discovering, observing, and tracking objects in orbit. In addition to those components, any simulation of the SSN must also model the tens of thousand of objects that are in orbit, their reflective properties, and their orbital motions. Thus, any simulation of the SSN necessarily involved mixed continuous and discrete modeling techniques. This leaves us with a choice of whether to model the system primarily as a continuous time-stepped model, with special logic for detecting discrete events when they happen (to within the accuracy of the time step chosen), or to model it primarily as an event-driven system, dropping into continuous methods between events when necessary. We believe the appropriate approach is the latter, so that the global logic of the TESSA simulator is that of a DES, but some event methods include with them some time-stepped continuous integration, for example, for force-model orbital calculations.

The radars, telescopes and databases in the system are naturally modeled as discrete objects, and their behaviors of interest are modeled as *discontinuous* actions. At certain moments they point in the proper direction and take a measurement or image. These sensing actions are modeled as taking zero simulation time, or more precisely, they are modeled as taking place in one instant at the moment the measurement is finished. Other discrete objects in the model represent signal and image processing processes, cataloguing processes, and scheduling processes. Each of their events is also modeled as happening in a single instant at the moment it is completed.

The major continuous behaviors that have to be modeled are the orbital motions of the satellites and space junk and debris objects that the SSN sensors are tracking. However, the amount of computation time required to handle that is relatively small compared to that needed to accurately simulate the sensors themselves, and much of it can be done offline outside of the simulation proper, so that even with tens of thousands of orbiting objects the simulation as a whole is dominated by the discrete event part of the simulation. Note that it is very simple to parallelize the computation of the orbits of the objects being tracked, because to very high precision the objects do not interact at all and their orbits are all perfectly independent of one another, and the same exact code can be used for all of them

so that it is even appropriate for SIMD computation on a GPU (an improvement we hope to implement in the future). It is not nearly so simple to parallelize the simulation of the sensors and other major activities of the simulation, because they do interact directly and indirectly. Thus we conclude it makes sense to structure the SSA simulation as a whole as a discrete event simulation in a way that best captures the parallelism of the difficult-to-parallelize parts of the model, the event-driven parts.

Fig. 3 shows a diagram of a typical TESSA simulation [1] of one small configuration of the SSN, illustrated at the object level. It shows ground-based telescope sensors only, although other runs include various radar-based sensors as well. Most of the objects are individual processes (as recognized by the operating system) but each telescope object is actually represented by eight or more processes spread across the same number of processor cores, so there is considerable parallelism within the event methods of the telescope objects. Some of the other objects have threaded event methods as well so that there is additional opportunity for intra-object parallelism. The arcs in the graph represent event message communication "channels", indicating which objects send event messages to which others. The black arcs represent channels used in most SSN simulation configurations. The red arcs represent channels that allow for various kinds of scheduling feedback in the model so that, for example, the results of one telescope observation might influence which other observations are scheduled for later. It will require additional work before models with such feedbacks performs well because, as we explain later, they require the model code to advise the simulator of *lookahead* information, and currently the code for the SSN simulation objects do not yet do that.

Another reason to choose discrete event simulation as the main simulation paradigm is that the SSN is inherently multiscale and irregular in time. Event chains in a TESSA simulation of the SSN happen at time scales varying over 7 orders of magnitude. Some causal chains have events hours apart (catalog operations) and others with sub-millisecond time deltas (radar observations). This would make it difficult to choose a global time delta value in a classic time-stepped model, since generally speaking one is more or less forced to choose the smallest delta that is needed anywhere in the model, and in parallel time stepped algorithms that leads to tremendous waste of processor resources and large synchronization overheads. We note also that actual moments in simulation time when events happen in an SSA simulation are *not regularly spaced in time* and are *not statically determined*. Time stepping therefore is not really a reasonable control structure for the simulation. While time stepping algorithms can be dynamically adjusted to increase or decrease the time step occasionally, it cannot be done very often at large parallel scale because each change in delta requires a communication broadcast and also because it generally leads to a very unbalanced computation. Discrete event simulations, by contrast, can be composed of processes occurring at drastically different time scales without causing any technical difficulty, since a discrete event simulation can be arbitrarily irregular in its inter-event times with no added overhead penalties or inherent waste of processor cycles.

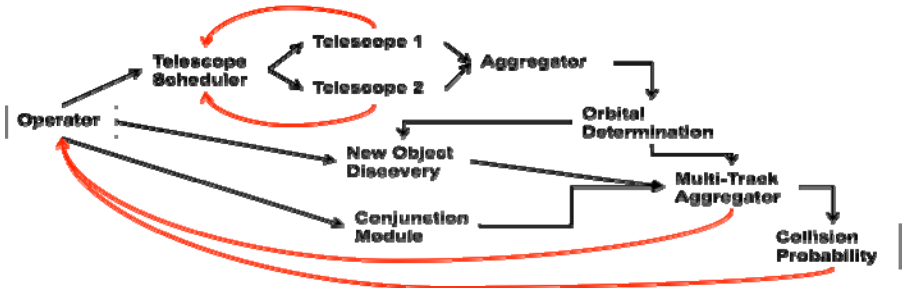

Fig. 3: Example of a TESSA simulation of the SSN observing debris, detecting new orbiting objects, calculating probabilities for collision with other space assets, and providing feedback to the operator of the network. The black connections show our current communication graph. The red edges represent potential feedback channels to be added in the near future.

Finally, we note that our SSA model had to be constructed from separately developed object models, each built by a different team of subject matter experts (optical, radar, orbital mechanics, etc.) It is generally hard to integrate several separately developed time stepped models into one federated model because the developers typically choose different and incommensurate choices of time step length. Even when they are constrained to be simple multiples of one another, the choice of global time step to be used for the combined model as a whole usually needs to be at least as small as the smallest one used in any component, and this causes huge computational waste and overhead in the parts of the simulation that do not need such fine time scales. There are, of course, ways around this, but they break the simplicity of the time stepping control structure, and in the limit approach that of discrete event simulation anyway.

3. PARALLEL DISCRETE EVENT SIMULATION: BUY OR BUILD? CONSERVATIVE OR OPTIMISTIC?

Having decided that the TESSA simulator had to be discrete event, and also parallel because of the size of the SSN simulations we were contemplating, we then had to decide whether to build our own simulator or port one of the several off-the-shelf ones available, and also whether to use a *conservative* or *optimistic* algorithm. These decisions were effectively forced on us by two key technical constraints: some of the telescope objects in our simulation both have a very large memory footprint and also require internal parallelism.

Normally one would almost automatically choose to adopt one of the dozen or so scalable parallel discrete event simulators that are available commercially or from other researchers. But essentially all off-the-shelf simulators are designed to only handle objects that are moderate in size (typically less than a few megabytes for optimistic simulators, or any size that will fit RAM for conservative simulators, e.g. up to 3 GB on 32-bit machines). They also generally assume that the event methods in each object are sequential, and executed within a single process (in the operating system sense) residing on a single node of a scalable cluster. But in the case of our TESSA SSN simulation these assumptions have to be overcome. The telescope objects in our SSN simulation will eventually have to create high resolution sky images that may be up to 24 GB in size —too large to fit in the memory of one node in most machines. Furthermore, the computation that takes place in a single event for such a telescope—the construction of a high resolution sky image as would be produced by a real telescope pointed in particular direction from a particular location at a particular moment—is so large that it requires internal parallelism or else the telescope objects will be come a severe performance bottleneck in the whole parallel simulation. We thus needed a simulator that could represent a *single object* (a telescope) as a collection of processes spread across several nodes of a cluster. That meant we had to build the simulator ourselves.

In choosing whether to build an optimistic or conservative simulator, we again had little choice. In order to understand this, a little background is in order. Space does not permit a comprehensive explanation of the differences between the two paradigms—for more information see [2]. But generally we can give a sketch of the differences.

Conservative PDES mechanisms use conventional synchronization primitives (process blocking and waking) to assure that events are executed only in increasing time stamp event order at each object. They assume that the execution of an event method is an *irreversible* action, and thus the simulator must execute all events in any one object in strictly increasing timestamp order (barring ties) and *never* execute two events in the same object out of order object because there is no way to correct a mistaken out-of-order execution. Instead, objects (processes) must be blocked as long as there is any doubt regarding which event message is the next one to be executed at each object. It is not sufficient just to sort the event messages that have arrived already and always process the one with the lowest time stamp, because a later-arriving event message might have a lower time stamp than any that is already enqueued, and if so it would be an error to execute any more events until that message with lowest time stamp arrives. The essential problem that characterizes conservative methods is to *recognize* that event message when it does arrive—to *know* that no later arriving event message for this object will ever have a lower time stamp than the ones we have in hand, in which case the simulator can proceed to process that event.

The additional information required to definitely recognize when an event in a conservative PDES can be safely executed and avoid deadlock or excessively slow execution is called *lookahead* information. Lookahead information is generally of the form "Object a will never again send an event message to object b with a time stamp lower than t". This is information that the simulator cannot determine on its own—it must come from the code of

the objects comprising the simulation. It is an extra burden on modelers and the authors of simulations to provide lookahead information to the simulator, but it is essential for conservative simulators.

Optimistic simulators are quite different. They assume that event method executions are in fact *reversible*, i.e. their side effects can be reversed by state-saving and -restoration[3], or by more complex methods such as reverse computation[4]. With that capability an optimistic simulator can use the much more powerful synchronization primitive of process rollback instead of process blocking. This allows the simulator to speculatively execute events without worrying about whether they are out of order, and to correct the problem later by unexecuting events that were out of order and re-executing them in the correct order. Additional parallelism can be achieved by allowing execution both forward and backward in time, provided the overhead is not too high. Surprisingly, this is an extremely effective and efficient way to perform parallel discrete event simulations, and it has the additional virtue of not requiring the simulation programmer to provide lookahead information. Optimistic simulators thus give the writers of simulation codes a major advantage.

We would have been happy to use an off the shelf parallel discrete event simulator such as ROSS[9] or SPEEDES[10] to run TESSA simulations of the SSN if it were possible. But they all have one major limitation which precludes their use for our purposes: none of them support multiple-process objects as required for representing telescopes in our SSN simulations, nor can they generally handle threaded events. Thus we were essentially forced to implement our own. We might have chosen to implement an optimistic simulator that supported synchronized multi-process rollback, but at the start of a major new simulation project we were in no position to take the time to do the research required and then to build a new kind of parallel simulator. So we decided to instead to take on a much more manageable project and build a conservative simulator that supports multi-process objects instead.

4. THE ARCHITECTURE OF THE TESSA SIMULATOR

At the time we needed to start building the TESSA simulator (March, 2008) we did not know much about how the simulation of the SSN would evolve, and we had only the crudest estimates of the scale and performance that the eventual simulation would require. (Our estimates turned out to be low.) We also had to build the first version of the simulator quickly, so that it was operational in a matter of a few weeks and could be used to produce a movie to meet an early milestone and to demonstrate our capability to potential sponsors.

In order to build the simulator quickly we decided to use software elements we had on hand. A key software layer we rely on is Co-op [11], a parallel components framework built a few years earlier that was not designed with discrete event simulation in mind, but happened to have a number of properties that make it useful for building the TESSA simulator:

- Co-op allows the components of an application to be treated as objects in an object oriented framework, with components (objects) having references to one another and being able to apply methods on one another using *remote method invocation (RMI)*. We use Co-op objects to represent objects in the TESSA simulation of the SSN, and Co-op methods to represent event methods on those objects.

- Co-op components (objects) are actually arrays of processes potentially spread across multiple nodes of the underlying cluster. This easily solved the problem of allowing a single telescope object to be spread over multiple nodes and to allow its methods to achieve the required internal parallelism from multiple processes. Remote Method Invocations on multi-process objects work the same without regard to whether the object is a single process or an array of them.

- RMI could be used as the means to transmit and enqueue event messages, or lookahead messages.

- Co-op allows dynamic allocation and deallocation of processor nodes and dynamic launching and termination of components, which in turn directly allows us to permit the dynamic creation and destruction of objects in a TESSA simulation. This has proved convenient for the launching and initialization of the objects in a complex simulation before it begins actual execution.

- Co-op components and their methods can be freely written in any of six different programming languages (C, C++, Fortran 77, Fortran 90, Python, and Java) because of inclusion of the Babel[8] language

interoperability tool. This is significant because it allows different expert teams to code different objects in the simulation in a language that is comfortable to them, without the need for to require a common language throughout the simulation.

We made one key design decision that simplified the initial construction of both the TESSA simulator and the SSN simulations run under it. We did not permit any feedback, i.e. any cycles in the communication graph, of SSN models to be run under TESSA. This allowed us to avoid runtime deadlock, and to delay consideration of which of the many lookahead algorithms we might use. It also relieved the model builders of the need to calculate lookahead information at the early stages of the project when many design decisions were up in the air and not everyone understood how PDES worked. Fortunately the early SSN models we needed to build did not require feedback. We are just now correcting this limitation, and will be building and running models with complex feedback in the future.

5. CURRENT STATUS OF THE TESSA SIMULATOR

The TESSA simulator has been running now for two years. A typical large run contains 20 to 30 objects (radars, telescopes, aggregators, conjunction detectors, collision probability calculators, catalog objects, and a few others). Those 30-40 objects are spread over 100-200 cores (because of internal parallelism in some of the objects) and a dozen nodes. Demo runs typically take 6-12 hours, not including some precomputation (notably debris generation, which we do not yet do inline with the rest of the simulation) or postprocessing (e.g. for visualization).

Fig. 4 shows the processor utilization across all nodes in one typical, non-demo run of a TESSA simulation. In this case we were simulating a 6-telescope SSN that was tracking the FORTE satellite. We were using 64 processors (cores) for about 1 hour. During that time we achieved about 50% processor utilization, a rather high utilization as irregular parallel computations go. The non-idle time was dominated by the six telescopes objects that each do optical image generation followed by image processing to isolate streaks in the generated sky images that represent orbiting objects. Each of the six telescopes was represented as 8 processes, for a total of 48 cores.

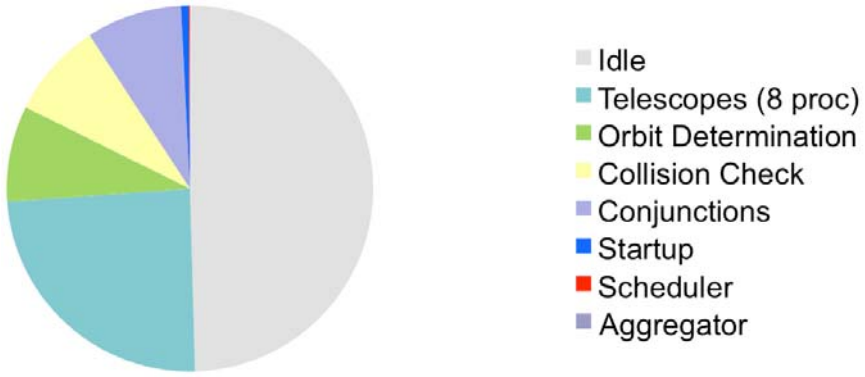

Fig. 4: Utilization of processor time in a TESSA simulation of 6 telescopes tracking the FORTE satellite.

This dominance explains the need for parallelism within the telescope/optical processing objects in the TESSA models to prevent them from becoming sequential bottlenecks in the simulation.

6. LIMITATIONS OF THE CURRENT TESSA SIMULATOR

While we have been developing TESSA and running simulations for nearly two years, TESSA is still limited in some ways. Many of these limitations are currently being fixed as we continue to develop, but one issue probably cannot be fixed with the current architecture.

Currently TESSA lacks any checkpoint/restart functionality. If a simulation fails, it must simply be restarted from the beginning. This has not been a problem in the past due to the modest scale and runtime of the problem so far. However, as we have begun to attempt more complex simulations the need for checkpoint/restart has become more

acute. A parallel checkpoint/restart framework is currently under development in the simulator, and will also require each object in the simulation to cooperate by offering methods to save and restore their own local states.

Another feature currently under development is the ability to run a TESSA simulation as a service under a Service Oriented Architecture (SOA). Launching and running any large-scale parallel application as a highly available service available is still a complicated development project, but is a goal for the next year.

The TESSA project currently lacks a full compliment of tools to analyze and optimize parallel simulation performance. We have raw performance instrumentation in the simulator, but analytical tools such as critical path calculation and performance visualization are still lacking. So far we have been successful without such tools, but as the simulations become more complex, more feedback intensive, and run for longer times and at larger scales, those tools will be essential.

TESSA has only recently been updated to offer APIs that simulations can use to provide lookahead information to the simulator, and it has only been demonstrated in very limited tests as of this writing. Our simulations of the SSN have heretofore been mostly cycle-free, so this was not an issue, but that is now changing. The simulation code has not yet been updated to provide the lookahead information to the simulator, although that will happen soon.

The largest software engineering limitation of TESSA is the non-portability of the software stack on which it relies, a consequence of short development time required at the very beginning of the project. In order to run on a cluster, TESSA requires Co-op, as indicated in Sec. 5 above, and Co-op in turn depends on a number of software services and packages that are not yet standard in high performance computing environments. One is Babel[8], a language interoperability and components package developed at LLNL and widely used elsewhere. Babel is quite portable, but is dauntingly complex to install in other environments. Co-op also currently requires a job launch mechanism that can launch multiple different codes inside of an allocation of processor nodes, and an MPI library that can support one MPI job launching other MPI jobs with a new, disjoint MPI_COMM_WORLD. Co-op also requires a full TCP/IP stack for the implementation of RMI between components. Co-op has thus far only been ported to Linux and AIX systems running either SLURM[5] or PBSPro[6], and with elan, mvapich[7], or IBM's MPI implementation. These software dependencies currently preclude running on some of the cutting-edge supercomputing architectures, such as IBM's Blue Gene line. The TESSA simulator's portability issues will probably not be resolved without a substantial redesign of Co-op and/or TESSA itself. It is thus doubtful that TESSA as currently implemented can be successfully ported outside of Lawrence Livermore National Laboratory without a major effort.

7. CONCLUSION

We believe that parallel discrete event simulation (PDES) is the fundamental simulation paradigm appropriate for modeling the Space Surveillance Network. We have built the TESSA simulator specifically to run SSN models, with a key capability that other PDES platforms do not support but that we found necessary: the ability to transparently support multi-process, multi-node objects with internally parallel events. The TESSA simulator is still a work in progress. We expect it to remain a conservative parallel discrete event simulator, but we intend over the next couple of years to remedy some of its deficiencies and extend its capabilities so that it can become a major tool in understanding how to use and improve the Space Surveillance Network.

ACKNOWLEDGEMENTS

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344

This is LLNL Information Management document number LLNL-CONF-454938.

REFERENCES

1. Olivier, S.S., "A Simulation and Modeling Framework for Space Situational Awareness," *Proceedings of the Advanced Maui Optical and Space Surveillance Technologies Conference*, Wailea, Maui, Hawaii, 2008.

2. Fujimoto, Richard, *Parallel and Distributed Simulation Systems*, Wiley-Interscience, January, 2000

3. Jefferson, David, "Virtual Time", ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 7, 3, pp. 404-425, July 1985

4. Carothers, C., Perumalla, K. S., and Fujimoto, R. M. "Efficient Optimistic Parallel Simulations using Reverse Computation", ACM Transactions on Computer Modeling and Simulations 9, 3, 224–253, 1999

5. Yoo, A., M. Jette, and M. Grondona, "SLURM: Simple Linux Utility for Resource Management, Job Scheduling Strategies for Parallel Processing", v. 2862 of *Lecture Notes in Computer Science*, pp 44-60, Springer-Verlag, 2003.

6. Jones, James Patton Jones (ed.), "PBS Pro 10.1 User Guide", Altair Grid Technologies, 2010.

7. Liu, J., J. Wu, and D. K. Panda, "High Performance RDMA-Based MPI Implementation over InfiniBand", *Int'l Journal of Parallel Programming*, 2004

8. Kumfert, G., J. Leek, and T. Epperly, "Babel remote method invocation" *Proceedings of the 21st International Parallel and Distributed Processing Symposium*, March 2007.

9. Carothers, C. D., Bauer, D., and S**.** Pearce , "Ross: a high-performance, low memory, modular time warp system", *Journal of Parallel and Distributed Computing*, v. 62 pp 1648-1669, 2002.

10. Metron, "SPEEDES: Synchronous Parallel Environment for Emulation and Discrete-Event Simulation", 2004, www.speedes.com

11. Jefferson, D., N. Barton, R. Becker, R. Hornung, J. Knap, G. Kumfert, J. Leek, J. May, P. Miller, J. Tannahill, "Overview of the Cooperative Parallel Programming Model", LLNL Tech Report UCRL-CONF-230029