Automated Load Balancing of a Missile Defense Simulation Using Domain Knowledge

Martin C. Carlisle
Department of Computer Science
United States Air Force Academy
2354 Fairchild Drive
USAFA, CO 80840-6234
carlislem@acm.org

Laurence D. Merkle[1]
Department of Computer Science and Software Engineering
Rose-Hulman Institute of Technology
5500 Wabash Avenue
Terre Haute, IN 47803
merkle@rose-hulman.edu

For discrete-event simulations with a large number of objects, parallelization provides an opportunity for improved performance. However, if simulation objects are not placed carefully, the additional communication costs can outweigh the performance gain from adding CPUs. For many simulations, the large number of objects makes manual placement of objects impractical. We present two new algorithms for static load balancing that, using a small amount of domain knowledge and run-time measurements, automatically discover objects of different classes that communicate frequently and place these objects on the same processor. We compare results obtained from a missile defense simulation implemented in SPEEDES using our algorithms to those obtained with previously published load-balancing algorithms.

Keywords: Static load balancing, missile defense, SPEEDES, domain knowledge, genetic algorithms

1. Introduction

Many simulation models require an unsatisfactory amount of time when run on a single processor. Parallelizing these simulations provides the requisite computational resources to achieve the desired running time, but also introduces communications costs that can outweigh the gains of parallelization. Load-balancing algorithms attempt to find the optimal tradeoff between placing simulation objects together to reduce communication costs and distributing them to gain parallelism. Static load-balancing algorithms assign simulation objects to processors before the simulation begins. Dynamic load-balancing algorithms use run-time information to change the allocation during the simulation.

Although it has been stated that it is preferable to use dynamic load balancing, static load-balancing schemes are attractive because of their simpler implementation and reduced cost at run time [1,2]. Furthermore, in certain cases static load-balancing schemes are competitive with dynamic load balancers [2]. Our simulation uses the SPEEDES framework [3], which uses an object-oriented computational model. A simulation model is created by defining simulation objects, object managers for them, and events. Object managers have the ability to control the placement of the simulation objects on processors; therefore, it is simple to implement a static load-balancing scheme, and previous research has found such schemes to be effective [4].

The simulation is of a single theater missile defense mission. Theater missile defense systems are deployed with troops to protect the troops from enemy missile attack. Theater missile defense systems [5] that are currently deployed and are represented in the simulation include the Patriot, Theater High Altitude Area Defense system (THAAD), Aegis, and others. Simulating these systems allows the DoD to develop strategies for the use of these systems without the expense of staging exercises with the real equipment, and to evaluate potential new missile defense acquisitions. The simulation in this work is actually used for joint exercises.

Figure 1 represents the simulated battlespace. The theater is divided into regions, with each region having its own battle manager, who is responsible for defending assets within the region. A theater manager coordinates the defense. This is important because a region that is under significant attack may need assistance from other regions between it and the attacker. For example, if the easternmost region in Figure 1 is under attack from an enemy to the west of the battlespace, the other three regions may have the opportunity to shoot down the incoming missiles.

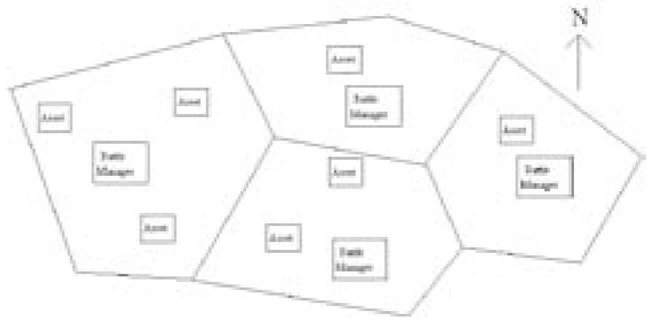

Figure 1. An example battlespace

Each battle manager has launcher(s) under its control (e.g., Patriot batteries). Each of those launchers has a certain number of interceptors. Incoming threats are referred to as missiles. Each of these real-world objects is modeled as an object in the simulation. Additionally, a simulation object, the ThreatFactory, is used to launch missiles, and other simulation objects perform I/O. Together, there are approximately 3,000 objects in the simulation, which simulates a 25-minute attack.

Although the simulation provides automated battle managers, it also allows humans to play these roles. The simulation has been modified to provide a "human-in-the-loop" mode, which throttles simulation time to real time. This distributed interactive simulation has a different goal for load balancing. When the computer is managing the battle, our goal is to minimize CPU time. On the other hand, when people play the part of the battle managers, our goal is to prevent the simulation from falling behind real-time (so that the players do not see delays while managing the battle).

The particular scenario used in this paper represents a desired wargame exercise for which the performance was unsatisfactory. There is continuous pressure to add more simulation objects to the simulation and increase the realism of the simulation. Previously, this simulation used the SPEEDES default layout of objects, a "card-dealing" algorithm where objects are distributed one at a time to each processor in turn. Occasionally, they would manually "stack the deck" to obtain better performance. This was a time-consuming process that would have to be redone for each new simulation, so it was desired to have an automated strategy for accomplishing this. We implemented two previous algorithms and designed two new algorithms for automated load balancing. Our algorithms use a small amount of domain knowledge to create an allocation. These algorithms require data from a benchmark simulation and use those results for subsequent runs.

Section 2 compares our work to previous algorithms for automated load balancing. Section 3 describes the two existing algorithms we implemented and the two new algorithms we devised for statically load balancing this simulation. Section 4 reports on the performance of these algorithms, using the goals of minimizing total simulation time and real-time lag. Finally, Section 5 provides conclusions and ideas for future work.

2. Related Work

Significant work has been done previously on the problem of load balancing objects in a simulation. One unique aspect of our work is that we are not only interested in possible speedup, but also the real-time lag of the simulation. The most closely related work was done by Wilson and Nicol [4]. They presented three algorithms for automatically allocating simulation objects to processors using the same simulation framework, SPEEDES. As with our work, they require data from a benchmark run of the simulation to create an allocation for subsequent runs. We implemented two of these algorithms and compare results of our new algorithms to these algorithms. We demonstrate that adding domain-specific knowledge to merge objects before performing the allocation generates improved performance.

Boukerche and Tropper [17] also perform a static partitioning of the simulation objects. They use a simulated annealing algorithm. This starts with an initial partition and refines that partition. One of the algorithms we propose uses a genetic algorithm to perform a similar refinement. Their work was done in the context of conservative simulations.

Gan, et al. [10] discussed using a combination of static and dynamic loadbalancing schemes. In particular, the static schemes they used were Metis [16] and Scotch [14]. Following Gan's approach, we peformed a graph partitioning using the simulation objects as nodes of the graph, and the edges weighted using various functions of the communications between those objects. Unfortunately, applying Gan's methods to our simulation did not provide an improvement over the default SPEEDES allocation. This may be a consequence of using an optimistic simulation engine, where rollbacks may have a cascading effect on the running time of the simulation, even though the communications that caused the rollback do not weigh heavily in the functions.

Vee and Hsu [15] also note the importance of preserving locality (i.e., minimizing the number of inter-processor messages) when performing load balancing. However, they assume that the simulation model has already been decomposed into a number of submodels, or logical processes (LPs), before their load-balancing strategy begins. Our scheme uses domain-specific knowledge to, in essence, create LPs which can then be load balanced.

Som and Sargent [9], although they focus on dynamic load balancing, also provide a pre-processing phase to combine simulation objects into what they refer to as *strong groups*. These strong groups are computed by examining a directed graph between simulation objects, where the edges are placed from an object to another when the first object places something in the queue of the second. They then remove some edges if they are deemed to be infrequent. Our research demonstrates that domain-specific knowledge can provide improved performance when strong groups cannot be determined automatically from the simulation.

3. Algorithms

Inputs for the load-balancing algorithms were obtained by tracing a representative simulation. As with all static load-balancing schemes, the quality of the output of the balancing algorithm is dependent on how representative the trace is. In our situation, we are guaranteed a very representative trace as the scenario for a human-in-the-loop wargaming exercise is decided in advance. To obtain this trace, the simulation was instrumented such that the following information is available for each committed event: the simulation object processing the event, the amount of CPU time consumed by the event, and the details of simulation events scheduled by the processing of this event. From this data, the following summary statistics were computed: the total CPU time consumed by each simulation object, the number of messages sent from each simulation object to every other simulation object, and the amount of CPU time consumed by each simulation object per five seconds of simulation time. The simulation models a 1,500 second battle; the five-second granularity was chosen to provide acceptable scheduling effectiveness while satisfying memory constraints.

Each algorithm described below takes as input the summary statistics, and produces as output a configuration file for SPEEDES with a complete allocation of simulation objects to processors. Since the file I/O of recording the trace data has a significant impact on the performance of the simulation (the trace files record hundreds of megabytes of data), tracing is disabled while allocations are being evaluated.

3.1 Lballoc1

Wilson and Nicol [4] proposed Lballoc1 as an algorithm for doing static load balancing while ignoring communication between simulation objects. Lballoc1 is a direct application of a bin-packing heuristic to load balancing, in which the processors are viewed as the bins into which the simulation objects must be packed. Specifically, each processor is viewed as a bin with unlimited capacity, and the size of each simulation object is its total measured CPU time. Thus, Lballoc1 considers each object in order of decreasing CPU time, and allocates it to the currently most lightly loaded processor.

Consider the following set of simulation objects:

ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Time	50	49	48	46	42	37	31	29	25	23	15	14	9	5	3	1

Lballoc allocates them in order of decreasing CPU time (left to right), looking only at the amount of CPU time currently allocated to each processor and selecting the smallest. So, on four processors, the final allocation would be:

ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
CPU	1	2	3	4	4	3	2	1	1	2	3	4	3	4	2	1

The total CPU times allocated to the processors are: 105, 106, 109, 107.

3.2 Lballoc2

Lballoc2, also by Wilson and Nicol [4], is a more complicated algorithm that seeks to place objects that communicate frequently together on the same processor. The first portion of the algorithm arranges the objects into a linear chain. This chain is built by performing a stable marriage algorithm [6] repeatedly on the objects. The attraction of each object to every other is determined by the number of communications between them. Each application of the stable marriage algorithm reduces the number of simulation objects by half. Then, for the next application, the attractions are computed as the sum of communications between the objects making up the merged objects. After ceiling ($\log_2 n$) marriage phases, a single merged object is obtained. These merges can be viewed as creating a binary tree, where each original object is at a leaf, and then the linear chain is obtained by a simple depth-first traversal. An allocation to p processors is obtained by splitting the chain at the p-1 locations that most evenly balance the load across the processors.

The following example demonstrates the design of Lballoc2. Consider the tree of simulation objects given in Figure 2.

The first level of the tree would occur if the pair of objects that communicates most are 8 and 2, the next pair (not including 8 and 2) is 4 and 1, etc. Then in the next phase pairs of simulation objects are considered (so the pair {8,2} communicates most with the pair {4,1}). The process repeats until there is only a single simulation object. An in-order traversal of the tree creates an ordering of the objects, {8,2,4,1,5,7,6,3}, and this ordering is partitioned across the processors to minimize the amount of work on the most heavily loaded processor.

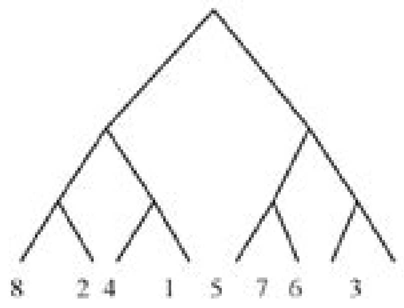

Figure 2. A tree of simulation objects

3.3 Adding Domain-Specific Knowledge

In the missile defense simulation, the battle manager objects perform the vast majority of communications. Each battle manager must coordinate its actions with and report its status to all of the other battle managers. This coordination generates a large number of messages. These large numbers of communications between battle managers mask the optimizations that can be attained by co-locating the battle managers with other types of objects, e.g., the interceptor launchers associated with the managers. Lballoc2 misses these optimizations. For large simulations, it may often be the case that there are groups of objects of different types that are communicate frequently, and correspondingly should be co-located in the simulation to obtain the best performance.

We allow the user to specify domain-specific knowledge about the relationships of objects of different classes. For an object class, we can specify that each object of that class should be merged with an object of a different class. For example, in our simulation, we specified that each launcher should be merged with a battle manager. We can also specify which direction of communication to consider. That is, we can either merge each launcher with the battle manager that sends the most messages to it, or we can merge each launcher with the battle manager to which it sends the most messages. In our simulation, we specified the following four merge phases:

1. Merge each Launcher with the Battle Manager to which it sends the most messages.
2. Merge each Sensor with the Battle Manager to which it sends the most messages.
3. Merge each Interceptor with the Launcher that sends it the most messages.
4. Merge each Missile with the Interceptor that sends it the most messages.

Once the merges are complete, any load-balancing algorithm can be used to distribute the computation amongst the processors. We considered two such algorithms: the same bin-packing strategy as used in Lballoc1, and a genetic algorithm.

3.3.1 Bin-pack Merged

Bin-packing is an example of an NP-Hard problem, which unlike many others, has a heuristic solution that can be proven worst-case to be within a constant factor of two of the optimal (the number of bins used is never more than twice the optimal). Furthermore, empirical studies have shown that the best-fit decreasing heuristic generally produces good solutions, and compares favorably with other heuristics [7]. Thus, if the merges give us good communication across processors, combining this with bin-packing should provide a balanced approach. This algorithm performs the merge phases described above, sorts the merged objects by decreasing CPU time, and then allocates the objects in order to the least loaded processor.

3.3.2 Genetic Merged

Since objects in the simulation do not contribute to the computation load throughout the simulation time (e.g., an interceptor only requires CPU time for the interval in which it is in flight), it seemed possible that an allocation that balanced the total load might inadvertently schedule too much work in particular time intervals. For example, consider the case where there are 25 simulation objects, where each simulation object is active for only a single second of the five seconds of simulation time. Suppose the first five objects are active for the first second, the second five the second second, etc. Considering only the total load, a perfect balance for five processors would be to place the first five on the first processor, the second five on the second processor, etc. Unfortunately, in this case we actually achieve no parallelism, since only one processor is active for each five seconds of simulation time. To avoid this, we designed a genetic algorithm (GA) that considered CPU time across intervals of five seconds of simulation time. Genetic algorithms (GAs) are a form of computation inspired by theories of evolution. This places them in the class of algorithms called Evolutionary Algorithms (EAs). In a landmark paper [11], Bäck and Schwefel provide an excellent review of evolutionary algorithms, including a historical perspective and a formal definition attempting to unify them. Merkle and Lamont generalized the definition to provide a more rigorous version that precisely captures the essential nature of EAs [12].

In GAs, data structures called *individuals* are used to represent possible solutions to a problem. In our problem, each individual is a possible allocation. This data structure is simply an array indicating, for each object, to which processor it is assigned. With a large number of simulation objects, this makes for quite large individuals in the GA, but this had no noticeable effect on the effectiveness or efficiency of the algorithm on the problem instances of interest.

The most distinguishing characteristic of GAs is that they manipulate collections of individuals, called *populations*, and that the operations performed on each individual may depend on the other individuals in the population. This is because the operations are inspired by the concepts of the theory of natural evolution, including fitness, selection, mutation, and recombination.

The concepts of fitness and selection are closely related, as are the fitness functions and selection operators that are inspired by those concepts and used in GAs. The *fitness function* assigns each individual a "fitness" based on some evaluation of the solution that it represents. In the case of optimization problems, which are perhaps the most common application area for GAs, the fitness function is related somehow to the objective function of the problem. The specific relationship between an individual and the objective function is one of the design issues involved in applying GAs. Once the fitness of each individual has been assigned, a *selection operator* randomly selects individuals from the current population to copy into the next population. More fit individuals have higher probabilities of selection. Early GAs used fitness-proportionate selection operators, meaning that the expected number of copies of each individual in the next population was proportional to its fitness (normalized by the average fitness). This approach has a number of theoretical shortcomings, so few modern GAs use it. Instead, they use selection operators for which the behavior is invariant to scaling and translation of the fitness function. The selection operator used in this research, binary tournament selection, is both scale and translation invariant and is used fairly commonly.

Mutation operators randomly alter individuals as a means of exploring the search space in neighborhoods of known good solutions. In the case of GAs, it is common to "flip" each bit in the string with a probability called the *mutation rate*. Traditionally this probability is quite small, with the intent of investing a small amount of computational resources in searching locally in the vicinity of solutions that are presumably relatively good. However, it is now commonly recognized that for some applications, including the one described in this article, greater effectiveness can be achieved by using non-traditional (i.e., larger) mutation rates. It is also fairly common to incorporate domain knowledge in the design of specialized mutation operators; we use a specialized mutation operator rather than simply flipping bits in the binary representation of individuals.

As mentioned above, the feature of genetic algorithms that most clearly distinguishes them from related algorithms is their manipulation of collections of individuals. This manipulation occurs in the form of *recombination operators*, which randomly select features from two or more individuals to create new individuals. For GAs with fixed-length binary string representations, recombination operators are usually called "crossover," and there are several common variations. The earliest and easiest to understand is *single-point crossover*, which randomly chooses a *crossover point* within the length of the two individuals being recombined, and exchanges the parts of the individuals following the crossover point. *Two-point crossover* chooses two crossover points, and exchanges the parts of the individuals between the two points. *Multi-point crossover* extends this idea to more than two crossover points. As with mutation operators, it is common to incorporate domain knowledge in the design of specialized crossover operators. The research described in this article is an example of that approach.

In successful applications, the combined effect of the selection, mutation, and recombination operators is to gradually produce populations of individuals that represent very good solutions to the underlying problem, in analogy to the principle of "survival of the fittest." As previously mentioned, for each simulation object, we computed how much CPU time it used during each five seconds of simulation time. For our GA's fitness function, we first determined the amount of CPU time allocated to the most heavily loaded CPU during each five seconds of simulation time. We then added these together across all of the five-second intervals. The fitness function was a weighted sum of this time total with the number of off-processor messages generated by the allocation.

In our GA, each individual represented a possible allocation, and each population contained 100 individuals. The initial population consisted of randomly generated individuals. A modified binary tournament selection was used to propagate the fittest individuals. Specifically, the individuals were randomly paired, and one round of a tournament was used to generate 50 individuals for this generation. Then, the original individuals were randomly paired again, and another round of a tournament selected the second 50 individuals. Note this guarantees the fittest individual of the previous generation appears twice in this generation (once from each tournament). Other individuals may appear 0, 1, or 2 times in the new generation (if there is a unique least fit individual, it is guaranteed not to appear).

For each individual (possible allocation) selected for mutation, we randomly redistributed the simulation objects between two of its processors. Specifically, for each simulation object assigned to one of these processors, we moved it to the other with probability one-half. Based on our experiments, we found a mutation rate of 30% to generate good results.

We used a domain-specific crossover operator. In each application of the operator, one parent acted first as a "donor" while the other acted as an "acceptor" to produce one offspring, and then the parents exchanged roles to produce a second offspring. The construction of each offspring occurred as follows:

1. Initialize the offspring by making a copy of the acceptor.
2. Determine the set S_B of objects allocated to the "best" processor by the donor and the set S_W of objects allocated to the "worst" processor by the acceptor. The determination was based on a weighted sum of two quantities: the difference between the CPU time allocated to that processor and the average CPU time, and the number of off-processor messages sent by objects allocated to that CPU.
3. Within the offspring, deallocate all of the objects in $S_B \cup S_W$ (necessarily leaving at least one processor of the copy empty).
4. Within the offspring, allocate all of the objects in S_B to an empty processor.
5. Within the offspring, allocate each of the objects in $S_W - S_B$ using the best-fit heuristic of Lballoc1.

Crossover was performed on 70% of the population. That is, 70% of the individuals were selected randomly and mated in pairs. These two parents generated two offspring using the method described above. Then, of these four (two parents and two offspring), the two best were reinserted into the population.

Experimentally, we found that the results converged after about 100 generations (it appeared to be approaching an asymptotic limit), so in each case, we allowed the genetic algorithm to run for 100 generations on the merged objects. Note that each new generation does not require an additional run of the simulation on the parallel computer. The genetic algorithm runs within a few minutes on a single processor (each generation required only a couple of seconds).

4. Results

We used three metrics to determine how well our allocations performed: wall-clock time, maximum real-time lag, and real-time integrated lag. The metrics presented represent the average of three runs of the simulation. Wall-clock time is obtained by simply letting the simulation run as fast as possible, and measuring the total elapsed time. Since our simulation could be run with people playing the roles of the battle managers, we ran the simulation again, throttled to real time. For these runs, we measured the maximum real-time lag (how far did the simulation fall behind real time at the worst point), and real-time integrated lag. Real-time integrated lag is designed to measure the total amount of lag in the simulation. Consider the plot of lag across real time given in Figure 3.

Real-time integrated lag is the sum of the lag areas beneath the x axis. The integral is estimated using the trapezoid rule given measurements that occur every second of real time. Each of these runs was performed on a 24 processor SGI Origin 3000 series supercomputer, running in isolation mode. We did not run SPEEDES on four of the processors so that display processes could run on dedicated processors. The display processes are not SPEEDES simulation objects, but instead constantly update the screen with information about the simulation. These processes are ignored by our algorithms.

Figures 4 and 5 show the As Fast As Possible (AFAP) simulation time for the four allocation schemes we implemented, as well as the results from using the default allocation provided by SPEEDES. Note that Lballoc2, while generally the worst of the allocation schemes we implemented, has the best performance on 20 processors. Overall the best performance occurs using the bin-pack merged scheme on 16 processors.

Figure 6 shows the maximum lag behind real-time in seconds. Generally the ranking of the algorithms is similar for maximum lag; however the best case now occurs on 12 processors using the bin-pack merged allocation.

Figures 7 and 8 provide the real-time integrated lag for the simulation runs. Again the ranking of the algorithms is similar, with the best results occurring on 12 processors using the bin-pack merged allocation.

The experimental results were encouraging. For each algorithm, the best performance obtained was better than the best performance of the SPEEDES default scheme on any number of processors. Furthermore, the two domain-specific algorithms outperformed their counterparts. Not only did the domain-specific

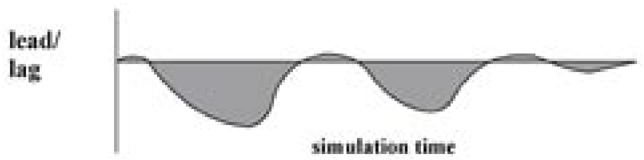

Figure 3. A sample lead/lag plot across a simulation

	As Fast As Possible Time (seconds)					
Algorithm\ #Proc	1	4	8	12	16	20
Default	2836.35	1088.97	755.25	702.35	719.03	808.54
lballoc1		1097.47	800.69	686.30	633.30	745.43
Lballoc2		1109.19	749.94	704.64	711.81	593.53
genetic merged		1014.82	702.37	651.68	633.97	715.53
bin-pack merged		1072.22	726.93	603.67	585.42	594.85

Figure 4. As Fast As Possible simulation time

algorithms have better performance, but they also obtained this performance using fewer processors.

Somewhat surprisingly, the bin-pack merged algorithm did better than the genetic merged algorithm. Analyzing the data, we note that the evaluation function for the genetic merged allocation was less than 0.5% better than that of bin-pack merged. We suspect that the bin-pack merged allocation performed better as it tended to cluster the low CPU objects together on lightly loaded processors, whereas the genetic allocations were more scattered. This is important because these low CPU objects, missiles and interceptors, publish their information via proxies to all of the battle managers. If they are on heavily loaded processors, they can force the other processors to keep rolling back. An initial test of this hypothesis was performed by taking the Lballoc2 allocation on 20 processors, and scrambling the allocation of the missiles for one test, and scrambling the allocation of the interceptors for another. Scrambling the allocation of the interceptors more than doubled the integrated lag (to 3533.79 from 1542.73). Scrambling the allocation of the missiles increased the integrated lag by over a factor of 10 (to 16466.55 from 1542.73). While these results are very preliminary, they suggest that missiles and interceptors, despite being both low CPU objects with few messages, have an important effect on the performance of the simulation.

5. Conclusions and Future Work

We have compared four static load-balancing schemes on a theater missile defense simulation, including two previously published static load-balancing schemes for SPEEDES. We have found that by adding a very small amount of domain-specific knowledge, we obtain significant performance improvement. The new algorithms reduce integrated lag by over 75% compared to the SPEEDES default allocation and over 10% compared to Lballoc2. They also reduce maximum lag by 62% compared to the default allocation and over 16% compared to Lballoc2. Finally, they reduce the total As Fast As Possible simulation time by over 16% and 1.3% compared to the default allocation and Lballoc2, respectively. Furthermore, the optimal results for the bin-pack merged scheme occur on fewer processors than Lballoc2.

Since the amount of information that must be provided by the domain expert is quite small, these algorithms could be readily applied to other simulations having multiple object classes. Using this small amount of information, these algorithms are able to automatically discover related objects of different classes and place them together.

As can be seen in the experimental results, this simulation scenario is still far from achieving perfect parallelism. Additionally, the real-time lag, while significantly improved, is not quite acceptable (ideally the maximum real-time lag won't exceed 10 seconds). For these reasons, and because we anticipate increased demand for the simulation (larger numbers of objects, more fidelity, etc.), we plan to explore other ways of improving efficiency, such as reducing the number and size of communications between objects. This will require analysis of the code for the simulation itself,

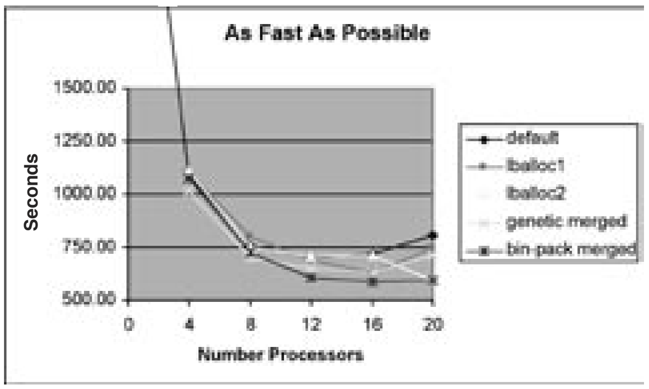

Figure 5. Graph of AFAP simulation time.

or the use of a different simulation framework, such as one of those proposed by Nutaro [18]. We expect the load-balancing schemes to work under any optimistic simulation framework, as they balance the load between processors, and reduce the number of output events to other processors. Additionally, we plan to explore load balancing in the context of the larger national missile defense scenario [8].

	Maximum Real-Time lag (seconds)			
Algorithm\ #Proc	8	12	16	20
Default	41.26	29.45	51.92	52.84
Lballoc1	70.51	20.12	18.77	31.87
Lballoc2	44.6	25.76	47.4	13.15
genetic merged	25.07	18.39	13.87	19.17
bin-pack merged	42.76	11.02	12.81	12.67

Figure 6. Maximum real-time lag in seconds

	Real-Time lag (seconds^2)			
Algorithm\ #Proc	8	12	16	20
Default	7682.18	5604.99	18807.43	12388.49
Lballoc1	13930.04	4130.61	3761.77	5670.99
Lballoc2	7494.57	4227.07	8338.64	1542.73
genetic merged	3823.65	3046.03	2607.07	4297.31
bin-pack merged	9999.13	1381.24	2384.19	2460.89

Figure 7. Real time integrated lag data

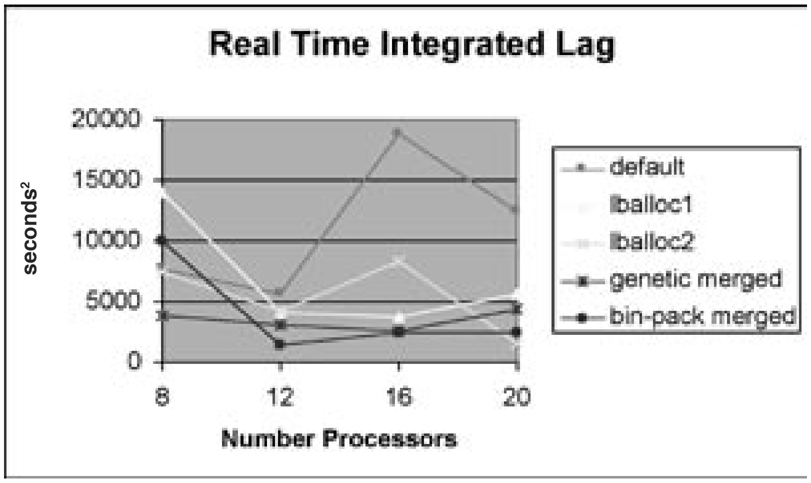

Figure 8. Integrated lag data

6. References

[1] Stone, H.S. 1977. Multiprocessor Scheduling with the Aid of Network Flow Algorithms. January 1977 *IEEE Transactions on Software Engineering*, Vol. SE-3, No. 1, 85-93.

[2] Sanders, P. On the Competitive Analysis of Randomized Static Load Balancing. Available at http://citeseer.nj.nec.com/79406.html.

[3] Steinman, J. 1992. SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete-Event Simulation. In *International Journal in Computer Simulation*, Vol. 2, No. 3, 251-286.

[4] Wilson, L.F., D.M. Nicol. 1995. Experiments in Automated Load Balancing. In NASA CR-198241, ICASE Report #95-80.

[5] Ballistic Missile Defense Organization. Theater Missile Defense Systems. Available at http://www.acq.osd.mil/bmdo/bmdolink/html/tmd.html.

[6] Sedgewick, R. 1988. *Algorithms, Second Edition.* Reading, MA: Addison-Wesley.

[7] Baase, S., A. Van Gelder. 2000. *Computer Algorithms: Introduction to Design and Analysis, Third Edition*. Reading, MA: Addison-Wesley.

[8] Ballistic Missile Defense Organization. National Missile Defense. Available at http://www.acq.osd.mil/bmdo/bmdolink/html/nmd.html.

[9] Som, T., R. Sargent. 2000 Model structure and load balancing in optimistic parallel discrete event simulation. In *Proceedings of the 2000 Workshop on Parallel and Distributed Simulation (PADS),* IEEE, 147-154.

[10] Gan, B. P., Y.H. Low, S.J. Turner, W. Cai, W.J. Hsu, S.Y. Huang. 2000. Load Balancing for Conservative Simulation on Shared Memory Multiprocessor Systems. *Proceedings of the 2000 Workshop on Parallel and Distributed Simulation (PADS),* IEEE, 139-146.

[11] Bäck, T., H.P. Schwefel. 1993. An Overview of Evolutionary Algorithms for Parameter Optimization. *Evolutionary Computation*, Vol. 1, No. 1, 1-23.

[12] Merkle, L.D., G.B. Lamont. 1997. A Random Function Based Framework for Evolutionary Algorithms. In *Proceedings of the Seventh International Conference on Genetic Algorithms*, Thomas Bäck (ed.), 105-112.

[13] Langdon, W. B., et al., editors. 2000. *Proceedings of the Genetic and Evolutionary Computation Conference*. San Francisco, CA: Morgan Kauffman Publishers.

[14] Pellegrini, F.. J. Roman. 1996. SCOTCH: A Software Package for Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In *Proceedings of HPCN '96*. Brussels, Belgium: LNCS 1067, Springer, 493-498.

[15] Vee, V.Y., W.J. Hus. 2000. Locality-preserving load-balancing mechanisms for synchronous simulations on shared-memory multiprocessors. *Proceedings of the 2000 Workshop on Parallel and Distributed Simulation (PADS),* IEEE, 131-138.

[16] Karypis, G. V. Kumar. 1995. Multilevel Graph Partition and Sparse Matrix Ordering. *International Conference on Parallel Processing*.

[17] Boukerche, A. C. Tropper. 1994. A Static Partitioning and Mapping Algorithm for Conservative Parallel Simulations. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS)*, IEEE, 164-172.

[18] Nutaro, J. 2003. *Parallel Discrete Event Simulation with Application to Continuous Systems*. Ph.D. Dissertation, University of Arizona.

Endnotes

[1]Work performed while this author was a member of the faculty at the U.S. Air Force Academy.