Optimizing Massively Parallel Simulations of Infection Spread Through Air-Travel for Policy Analysis

Ashok Srinivasan
Dept. of Computer Science
Florida State University
Tallahassee, USA
Email: asriniva@cs.fsu.edu

C.D. Sudheer
IBM Research
New Delhi, India
Email: sudheer.chunduri@in.ibm.com

Sirish Namilae
Aerospace Engineering Department
Embry-Riddle
Aeronautical University, USA
Email: namilaes@erau.edu

Abstract—**Project VIPRA [1] uses a new approach to modeling the potential spread of infections in airplanes, which involves tracking detailed movements of individual passengers. Inherent uncertainties are parameterized, and a parameter sweep carried out in this space to identify potential vulnerabilities. Simulation time is a major bottleneck for exploration of 'what-if' scenarios in a policy-making context under real-world time constraints. This paper identifies important bottlenecks to efficient computation: inefficiency in workflow, parallel IO, and load imbalance. Our solutions to the above problems include modifying the workflow, optimizing parallel IO, and a new scheme to predict computational time, which leads to efficient load balancing on fewer nodes than currently required. Our techniques reduce the computational time from several hours on 69,000 cores to around 20 minutes on around 39,000 cores on the Blue Waters machine for the same computation. The significance of this paper lies in identifying performance bottlenecks in this class of applications, which is crucial to public health, and presenting a solution that is effective in practice.**

I. Introduction

Air travel has been identified as a leading factor in the spread of several infections [2], [3], [4], [5], [6], [7] and this has motivated calls for limitations on air travel during the current Ebola outbreak. However, such limitations carry considerable economic and human costs. Consequently, it is necessary to evaluate the extent of impact of air travel on spread of Ebola and to also identify policy options that can mitigate its spread without major disruption to air travel.

Computer simulations play a crucial role in evaluating viable policy options and exploring potential consequences of decisions taken by policy makers. For simulations to be effective, they need to be able to provide insight into the consequences of different policy choices that decision makers may make, and produce results under the time constraints required for quick action. Unfortunately, conventional models for infection spread through air-travel are too coarse-scaled to suggest fine-tuned policies, because they are typically based on analysis of aggregate passenger data. These models cannot account for changes in passenger interaction patterns due to changes in policies or procedures, which in turn influence infection spread. In addition, this approach typically requires good data, and data are often scant during the initial stages of an infection.

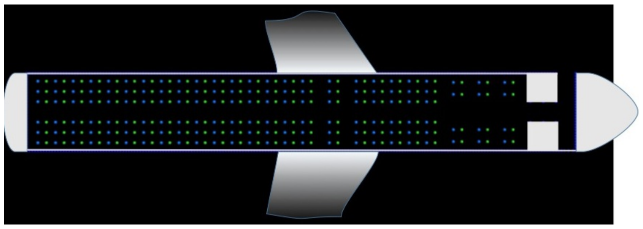
Fig. 1. Visualization of the initial state of a simulation

Project VIPRA takes a different approach, which involves tracking the trajectory of each individual passenger through the SPED model, which is a fine-scale causal model. (Figure 1 shows the initial state of a simulation of passengers deplaning.) The initial focus of this project is on Ebola, which is spread by contact with fluids of infected persons, either through direct contact or indirectly through contact with a common surface. We have shown that seating arrangement and boarding and disembarkation procedures play a major role in such contact.

Inherent uncertainties in human behavior make precise prediction of human movement or infection transmission difficult. This is magnified by the practical reality that models for all factors influencing an epidemic are not available, especially in the initial stages. Such sources of uncertainty are parameterized and our goal is to identify potential vulnerabilities in different policy or procedural choices across the parameter space. The goal is not to generate a single likely prediction but, rather, a set of possible scenarios, and to identify vulnerabilities due to any of these possible scenarios.

The large parameter space[1] due to the various sources of

[1]We use the term 'parameter category' for a type of parameter, such as passenger velocity in the absence of other passengers in the vicinity. For each parameter category, we may choose a range of values in order to deal with uncertainty in its exact value. The total number of parameters, which is referred to as 'parameter combinations' in certain contexts, is the product of the number of parameters for each parameter category. The large parameter space arises from a large number of parameter combinations.

uncertainty leads to high computational cost. This is a serious impediment to decision making, because such decisions are often taken during a course of a meeting, and it is necessary to produce results within the span of minutes to permit policy makers to explore different options. One approach to dealing with this bottleneck is to pre-compute results for possible policy options ahead of the meeting. The constraint on results within a few minutes can be relaxed with this approach. But one still needs results fast; models seldom yield reasonable results on a first attempt, and iterative refinement in required, especially as new data becomes available. Consequently, efficient computation, both in terms of time and computational resources used, is important even when results are pre-computed.

The focus of this paper is on the SPED model and its parallelization through parameter sweep on the Blue Waters machine, though we have used similar approaches also on BlueGene Q at IBM research and Stampede at TACC. We explain the computational structure of SPED and its workflow in Section II, along with a description of the Blue Waters system. In Section III, we show that optimizing the workflow of SPED can decrease the worst case time from a few a hours around 20 minutes for a 1331-core run.

In Section IV, we identify parallel file reads as a bottleneck in scaling the above computation from 1331 cores to around 69000 cores, which incurs time of around 45 minutes. We discuss two different optimization strategies, both of which reduce the time for this run to around 20 minutes. We then develop an algorithm for predicting the time taken by the computation and show that, with good load balancing, the same results can be obtained using 39,681 cores. We explain this algorithm in Section V.

In Section VI, we propose three load balancing strategies and evaluate their performance. We show that simple dynamic load balancing permits reducing the number of cores from around 69,000 to around 39,681 with around 4.2% increase in time. We also use a-posteriori bounds to show that this is close to optimal for a run using 39,681 cores, as the time taken is 13.6% higher than the optimum. Furthermore, we show that if we had a more accurate estimate of the time taken, then we can use alternate load balancing algorithms to maintain close to optimal time (differing by just 6%) while using only 32,405 cores.

We present our conclusions in Section VII and identify two directions for future work, in order to reduce the time taken and the amount of computational resources used. While the focus of this paper is on the SPED model, other applications of supercomputing for policy decisions can use a similar approach to dealing with uncertainty, and consequently, our approach has more generality than the specific application considered.

II. Computational structure and Computational Platform

A. *Computational Structure of SPED model*

The Self-Propelled Entity Dynamics (SPED) model has its origins in a code developed by co-author Namilae, while at Boeing, to simulate evacuation of airplanes. The parallel algorithm used is described in algorithm 1. Its primary computational structure is similar to Molecular Dynamics, with each passenger treated as a point particle that exerts a pair-wise repulsive force on other passengers if they come too close, based on a certain “potential“, whose parameters can be varied. Unlike with molecular dynamics, there is no attractive force; instead passengers come closer toward each other due to a forcing term. For example, movement toward the exit is the forcing term when simulating deplaning. A neighbor-list is maintained so that interaction is computed only for passengers within a certain cutoff. Apart from the above purely social-dynamics computation, certain behavioral characteristics are included to make the model realistic. For example, the manner in which passengers stop to collect luggage from overhead bins, move from interior seats to aisles, and move when not in proximity to other passengers are all modeled. The sources of uncertainty arise from both the social dynamics parameters and human behavioral parameters. In addition, SPED includes some randomness in the movement of passengers. Note that the VIPRA approach does not assume that the identity of an infected passenger is known. Rather, this is included as a source of uncertainty.

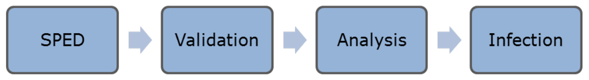

Fig. 2. Workflow

The SPED model does not directly simulate the movement of an infected person; analysis with such simulations will require a huge number of simulations to model all possible locations of the infected passenger. Instead, SPED generates trajectories for all passengers in a given simulation, with the simulation parameters being varied. A separate analysis phase analyzes each simulation. The analysis phase is fast, and can be used to identify contacts generated by infected passengers, which is then passed on to the infection model.[2] The workflow is shown in figure 2. This workflow includes a validation step before the analysis, for the following reason. While SPED desires to generate a wide variety of possible scenarios, including extreme cases that are rare, certain combinations of parameters lead to results that are clearly unrealistic. Such scenarios are removed during the validation phase.

We focus on simulations involving deplaning in this paper. The time taken by a computation, to a large extent, depends on the number of iterations required for passengers to deplane, because the simulation stops when all have deplaned. In addition, it also depends, to a lesser extent, on the number of

[2]Currently, Project VIPRA has not integrated the infection model with SPED; rather, it analyzes the total number of contacts and considers the maximum number of contacts under any scenario for a given policy or procedural choice, so that the one with the minimum such number may be considered ideal.

Algorithm 1 Parallel algorithm for the SPED model

```
1:  procedure
2:      Initialization
3:      for each process do
4:          Read parallel input (coordinates, input parameters)
5:          Loop over n time steps
6:          for each pedestrian do
7:              Compute desired velocity
8:              Compute forces
9:              Find new positions & velocities
10:             Compute averages, neighbors
11:         end for
12:         End loop
13:     end for
14:     Data collection
15: end procedure
```

pairwise interactions computed by the social dynamics force field.

B. Computational Platform

The above computations were performed on the Blue Waters machine at NCSA. We summarize the computational environment below.

The Blue Waters system is a Cray XE6/XK7 hybrid machine consisting of around 22,500 XE6 compute nodes all connected by the Cray Gemini torus interconnect. The XE6 dual-socket nodes are populated with 2 AMD Interlagos processors with a nominal clock speed of at least 2.3 GHz and 64 GB of physical memory. The file systems on Blue Waters are built with the Lustre file system technology. We used the IO profiling tool Darshan for analyzing the IO performance of the code. The default programming environment, Cray (PrgEnv-cray 5.2.40) compiler suite, was used for compiling the codes. The MPI library used is cray-mpich/7.2.0. The major code base is written in FORTRAN and it is provided with a C++ interface where MPI routines are used. MPI timer routines are used for timing the code. The timer is started right after MPI_Init and stopped just before MPI_Finalize. Thus, the times account for the total time of the simulation including IO. The compute nodes use a 64-bit Linux OS. The TORQUE job scheduler was used to submit the batch jobs. The Cray Application launcher (aprun) utility was used to launch applications on compute nodes.

III. Optimizing the Workflow

As in the development process of most models, after any refinements to SPED, simulations are first checked with a few parameters. Then, we increase it to 1331 parameters, and if results there appear reasonable, then perform a large simulations with around 69,000 parameters. The original code runs one process per parameter, and thus uses one core per process.

We first evaluated the ideal number of processes per node using the same parameter on each process for 10,000 iterations

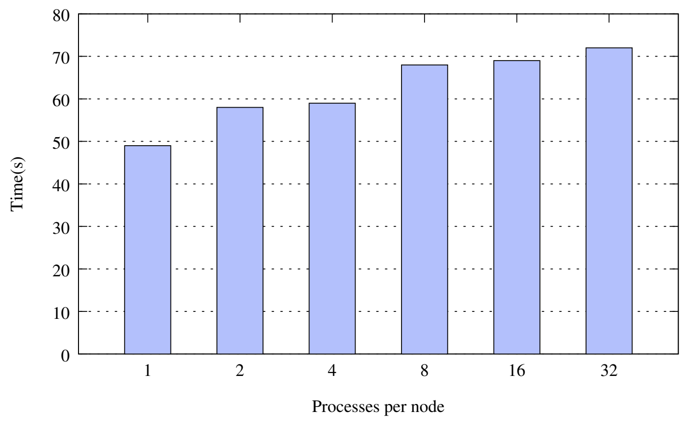

Fig. 3. Time for 10K iterations

(this number is not large enough for all passengers to have deplaned). Figure 3 shows the time taken as the number of processes per node are varied from 1 to 32. While one process per node clearly yields the best time, it is not realistic for a run needing 69,000 nodes; it would exceed the number of nodes available. A major drop in time taken is seen only at 4 processes per node. This is not realistic either; the queue wait time would be long and it would make inefficient use of the time allocation on the machine. Consequently, we choose the maximum of 32 processes per node. Note that the queue wait time typically increases as the number of nodes requested increases, and so it is not fruitful to decrease the number of processes per node for a small decrease in time. In addition, the charging algorithm for Blue Waters is based on the number of nodes used, and so it is preferable to use fewer nodes, as long as the computation time does not increase much.

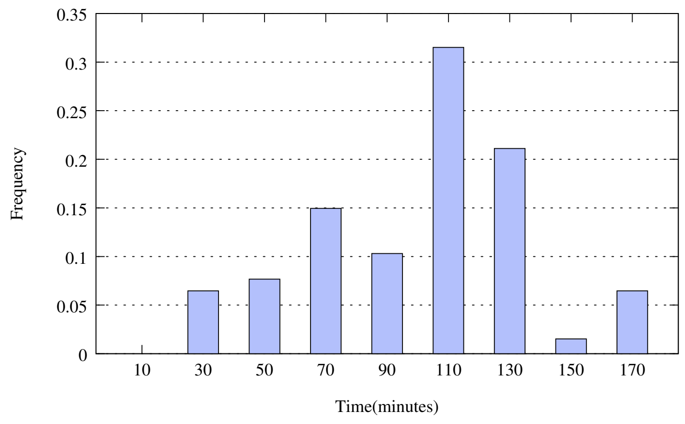

Fig. 4. Histogram of time taken for 1331 parameter combinations using the original code

Results from runs with a few parameters showed times of less than an hour. However, as seen in figure 4, the time taken for 1331 parameters is around 3 hours. We analyzed the cause for the poor performance. One of the causes was an inefficiency in a portion of code that was relevant for certain parameter values. This was easily corrected. The other cause

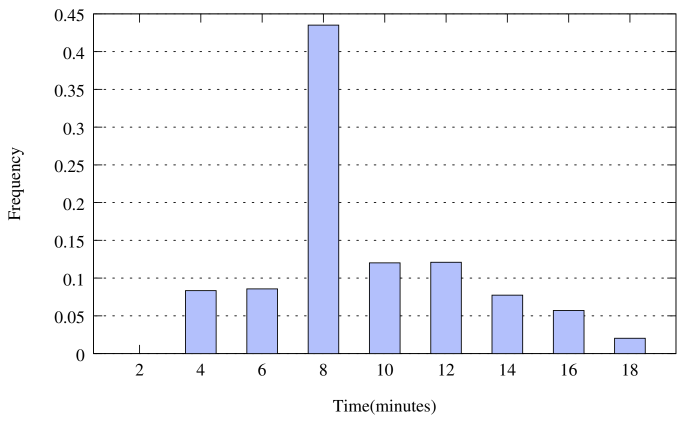

Fig. 5. Histogram of time taken for 1331 parameter combinations with first set of optimizations

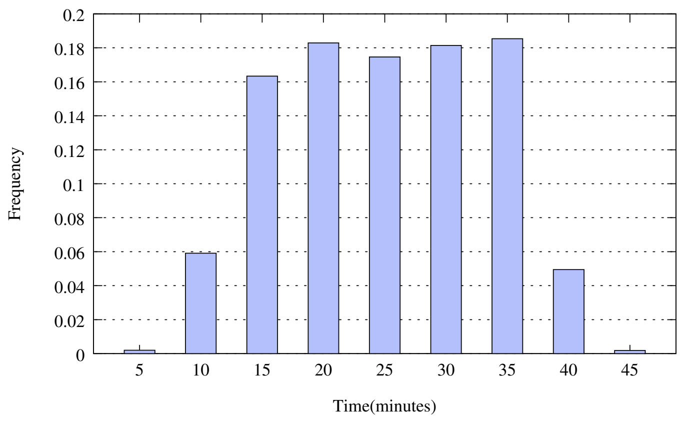

Fig. 6. Histogram of time taken for 68921 parameter combinations

was that long times were taken in certain simulations that the subsequent validation check showed as invalid. We changed the workflow (and SPED code) so that some validity checks are included in the course of the simulation and a simulation terminated when it is clear that it is not valid. Figure 5 shows time reduced to around 20 minutes, which is an order of magnitude improvement in performance.

IV. Parallel IO Optimization

We next performed a run with 69,000 parameters, which is our primary production run. Figure 6 shows that the time taken is around 45 minutes. We analyzed the cause of this performance drop.

As shown in algorithm 1, each process reads two input files (coordinates and parameters), which are of size a few KB each. We observed with the help of the Darshan [9] IO profiler that the reading of these files by all ranks is a bottleneck. The IO time involves the time for IO transfer and the time for metadata (open/seek, etc.). The file system on Blue Waters is a Lustre file system [11], and with large process counts (large number of files), metadata operations may hinder overall performance [10]. Figure 7 shows that the time for IO metadata is comparable to the time for the computation. The times for the metadata processing and other IO operations (reads/writes) is provided in figure 8.

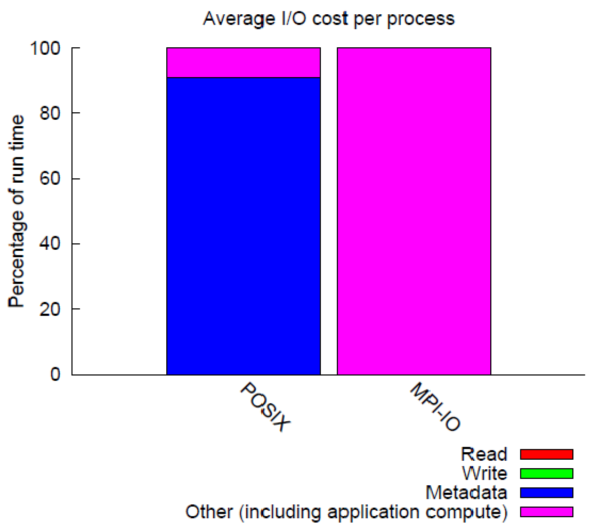

Fig. 7. IO profiling (before the IO optimization)

Average I/O per process

	Cumulative time spent in I/O functions (seconds)	Amount of I/O (MB)
Independent reads	0.000000	0.000000
Independent writes	0.010458	1.904802
Independent metadata	0.936319	N/A
Shared reads	0.000000	0.000000
Shared writes	0.000008	0.000013
Shared metadata	767.437892	N/A

Fig. 8. IO timings (before the IO optimization)

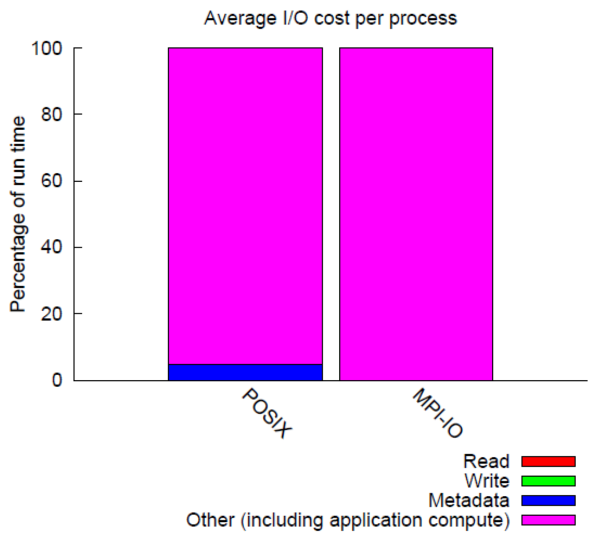

Fig. 9. IO profiling (after the IO optimization)

While the input files are shared across all the processes,

Average I/O per process	Cumulative time spent in I/O functions (seconds)	Amount of I/O (MB)
Independent reads	0.000000	0.000000
Independent writes	0.011489	1.916544
Independent metadata	2.741964	N/A
Shared reads	0.000000	0.000000
Shared writes	0.000000	0.000000
Shared metadata	6.817164	N/A

Fig. 10. IO timings (after the IO optimization)

each process periodically writes its output to an independent file. Lustre needs to maintain IO metadata information to enable shared file processing, while metadata overhead is not an issue with independent files. MPI IO and Lustre, therefore, effectively handle writing to the independent files. Consequently, IO write operations are not a bottleneck for this application.

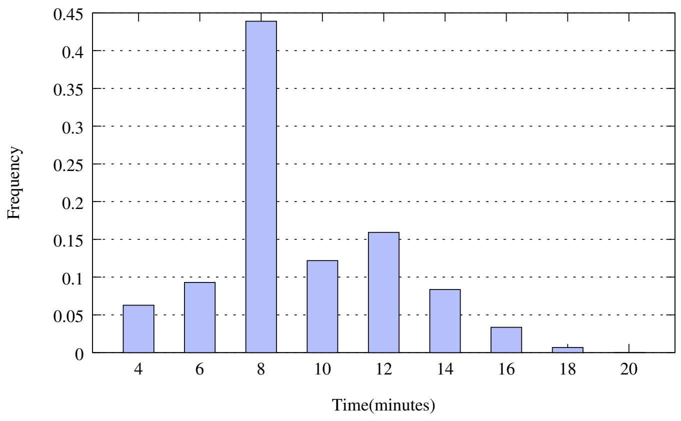

Fig. 11. Histogram of time taken for 68921 parameter combinations (after the IO optimization)

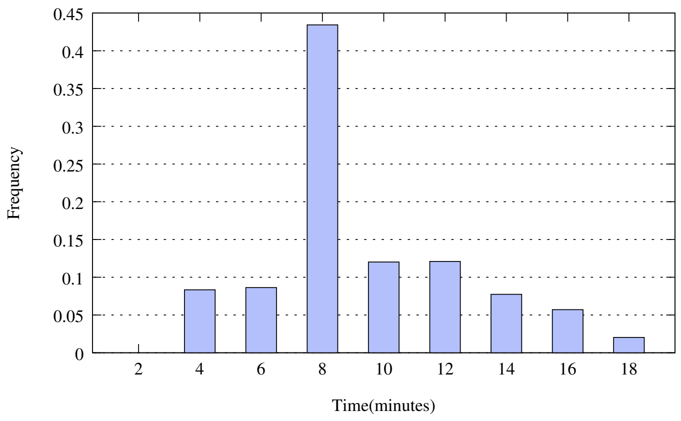

Fig. 12. Histogram of time taken for 1331 parameter combinations (after the IO optimization)

We consider two approaches to reducing the IO cost associated with reading common files. In the first approach, one process reads the two files and broadcasts their contents to all other processes. This results in a significantly reduced time spent with the IO metadata as can be seen from figures 9 and 10, thus improving the overall performance of the code by a factor of two.

In the second approach, we copy the files to the local memory of each compute node. Cray MPI Application launcher (aprun) has optimizations built in to enable efficient and scalable execution of certain commands, such as cp, in parallel[3]. The input files are just a few 100 KB, and so we place the files in /dev/shm file system.

We noticed that both the above approaches effectively handle the IO bottleneck. The first approach is more portable across different systems, whereas the second approach requires a memory based file system, such as /dev/shm or /tmp.

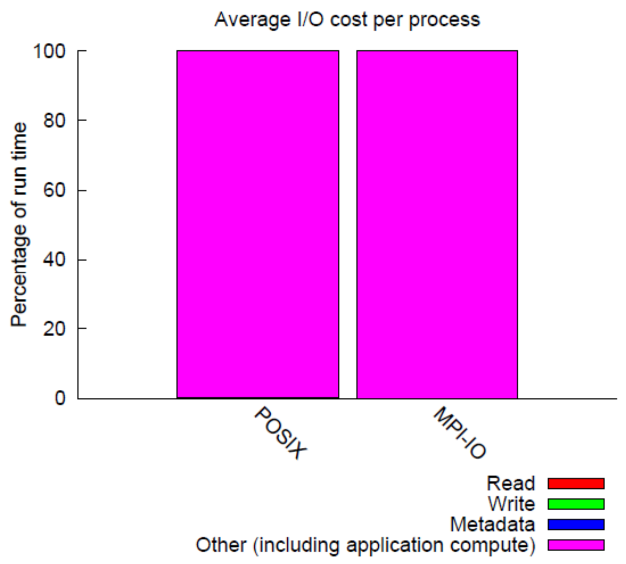

Fig. 13. IO profiling for 1331 parameters (after the IO optimization)

Average I/O per process	Cumulative time spent in I/O functions (seconds)	Amount of I/O (MB)
Independent reads	0.000061	0.000472
Independent writes	0.010548	1.889852
Independent metadata	0.097447	N/A
Shared reads	0.000000	0.000000
Shared writes	0.000000	0.000000
Shared metadata	0.000000	N/A

Fig. 14. IO timings for 1331 parameters (after IO optimization)

Figure 11 shows that the time taken for 69,000 parameters improves to around 20 minutes with this IO optimization. On the other hand, IO does not appear to be a bottleneck with the 1331 parameter problem as shown in figure 12. The histograms in both figures 5 and 12 are similar. This shows that IO metadata processing is not a bottleneck on 1331 cores.

[3]NCSA Blue Waters user support has given this input about Lustre setup on Blue Waters system.

Average I/O per process		
	Cumulative time spent in I/O functions (seconds)	Amount of I/O (MB)
Independent reads	0.000000	0.000000
Independent writes	0.010964	1.889852
Independent metadata	0.089385	N/A
Shared reads	0.000000	0.000000
Shared writes	0.000000	0.000000
Shared metadata	0.121298	N/A

Fig. 15. IO timings for 1331 parameters (before IO optimization)

The Darshan profile outputs for both the cases look similar to figure 13. The specific IO timings for the IO optimized case is given in figure 14. IO timings for the code before IO optimization, shown in figure 15, indicates some overhead in the metadata part. But, it is insignificant compared to the total time.

V. Determining Optimum Number of Cores to Use

The histogram for the above runs suggests that there is scope for reducing the number of cores used without increasing the time taken. Given a core count, this reduces to the problem of minimizing the makespan, provided we can estimate the time taken for each parameter choice.

Minimum Makespan is NP-hard, but several approximation algorithms exist [12]. The list-scheduling algorithm is equivalent to dynamic load balancing with tasks assigned in order to the next available core, and has an approximation factor of 2. If the processes are sorted by time before the above algorithm is applied, then the approximation factor is 4/3. This approximation bound can be improved to around 1.22 with the multifit algorithm [13] that repeatedly calls an approximation algorithm for bin-packing.

Algorithms with slightly lower bounds exist. However, they are more complex and reduce the approximation factor only marginally in practice [14]. In addition, post priori bounds on the factor 1.22 approximation for our problem show that it is close to optimum when the exact time is known.

Optimal	1015s
Multifit	1096s

TABLE I
1331 parameters on 721 cores.

Table I shows the time in seconds for the 1331 parameter problem using 721 cores. Time with multifit algorithm is close to the optimum, differing only by 7.9%. (We chose 721 cores based on theoretical bin-packing results that indicated this as the smallest count for which one could get the same time as using 1331 cores.)

The performances of the different algorithms for load balancing are compared in figure 16 for a run with 1331 parameters using 721 cores. Block mapping refers to assigning the first N/P parameters to the first core and so on (where N is the number of parameters and P is the number of cores). Striped mapping assigns the parameters to the cores one-by-one in turns; for example if there are 10 cores, than each core gets every tenth parameter. List Schedule (also referred to here as Plain Dynamic) is dynamic load balancing with a core getting the next available task once it is free.

The reordered mapping orders the tasks by their times in descending order before applying the list scheduling algorithm. The idea behind this strategy is that with the reordering the theoretical approximation factor is lowered from 2 to 4/3. Even though we don't have the exact time, we hope that our time prediction will lead to a better approximation.

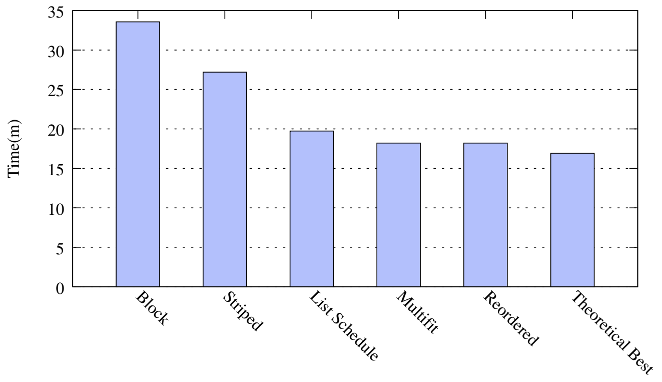

Fig. 16. Time taken with different mappings (block decomposition, striped decomposition, list schedule based) and make span scheduling for 1331 parameter combinations using 721 cores

The latter two approximation algorithms, which result in close to optimal performance, require estimates of the time taken to be used for load balancing. In contrast, simple dynamic load balancing, which assigns task to the next available core, automatically implements list scheduling. In all cases, though, one first needs to determine the number of cores to use.

This is the problem of bin-packing, which is NP-hard, and the dual of the makespan problem. We use the asymptotic factor 11/9 approximation algorithm for it [15]. Asymptotically better algorithms exist for bin-packing [16], [17]. However, they do not lead to a makespan algorithm that is significantly better in practice. Bin packing, of course, requires estimates of the time for each process and also the bin size. The smallest feasible bin size is the computation time of the parameter combination that takes maximum time. However, we also consider the tradeoff of slight increase in time, if it would lead to a substantial reduction in the number of cores needed.

We now consider the matter of predicting the time. We use the observation that the normal procedure is to perform a smaller run before the production run, in order to verify that the model and implementation work reasonably before expending computational resources. We developed an efficient algorithm for predicting the time.

This algorithm is based on the observation that the parameter sweep is performed on a lattice in both the trial run and the production run, with the dimension of the lattice being the number of parameter categories. We develop a few simple models for the time taken. We divide the parameter space into cells defined by the lattice of parameters of the trial run. Given a parameter combination, we identify its corresponding cell and evaluate its predicted time using one of the following models. The models are linear fit, quadratic fit, and maximum fit. In 3-dimensions, given parameter (x, y, z) linear fit predicts time as: $a + bc + cy + dz$, quadratic fit as $a+bx+cy+dz+exy+fez+gxz$, and maximum as the maximum of the time taken of the corner points. The maximum time is pre-computed for each cell.

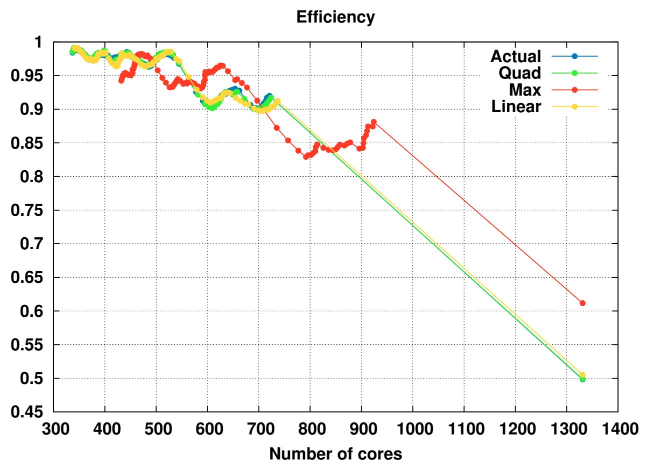

Fig. 17. Cores used vs. parallel efficiency with observed and predicted times for 1331 parameter combinations

The coefficients for each cell in the other two models are also pre-computed using a least squares fit on the parameters for the trial run. An efficient implement of least squares is developed for this problem as follows. We normalize a given cell to be in $[0, 1]^d$ in d-dimensions ($d = 3$) in the case studied in this paper. For any cell, the least squares algorithm requires the basis functions to be evaluated at the known points, which are the cell corners, to create the least squares matrix A. The above normalization leads to identical A matrices for all cells. If $A = QR$ is the QR factorization of A, and b the compute-time at the cell corners (which will vary with each cell), then the least squares solution for the coefficients is given by $R^{-1}Q^T b$. $R^{-1}Q^T$ is identical for all cells (and a small matrix, given that each cell has only 8 corners in 3-D) and is precomputed. The coefficients for each cell are, thus, computed as a small matrix-vector product. This solution is scalable as the number of parameters increases, though the matrix size does increase with the number of parameter categories.

We next evaluated the three models on the original problem (because, in a practical situation, times for the production run will not be available). We first predict the times for 1331 parameter problem based on the observed times for this same problem. We then predict the optimum number of cores based on these times. Figure 17 shows the comparison of the prediction with actual observed times and the three prediction methods. Quadratic and linear prediction were roughly equally good, though quadratic was slightly better. So, we use this in subsequent load balancing algorithms, except when we mention otherwise.

We next use results from 1331 parameter problem to predict times on 68921 parameter problem. Figure 18 shows parallel efficiency with different number of total cores used for both the exact times and predicted (quadratic fit) times for the 68921 parameter problem. Note that even though we show exact results with 68921 cores for the purpose of comparing, in reality this result is not available before the production run. The predicted times indicate that 39,000 cores can yield the same time as using 68921 cores if the load is balanced well. However, the actual times indicate the optimum number of cores to be around 32000 to yield the same time as using 68921 cores. However, even a reduction to around 39000 cores leads to a substantial improvement in efficiency.

Figure 19 shows the time taken with different number of cores used for both the exact times and the predicted times. The inference that can be made from figures 18 and 19 is that with a slight sacrifice on the total simulation time, a higher parallel efficiency can be obtained.

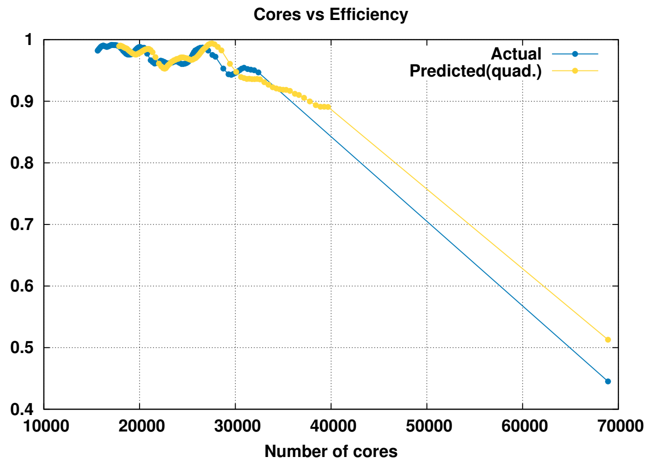

Fig. 18. Number of cores vs efficiency for 68921 parameter combinations using actual and predicted times

VI. Load Balancing

We next evaluate the different load balancing strategies on the 68921 parameter problem run on the estimated ideal number of cores.

Figure 20 shows timings on 39681 cores with plain dynamic load balancing, dynamic load balancing with the task list sorted based on times, and multi-fit based static mapping. The mappings for the later two cases is developed based on the predicted times. The timings of these three mappings are compared with the time taken when all the 68921 cores are used, which results in a poor parallel efficiency of 51%. The plain dynamic load balancing using 39681 cores takes time close to that with 68921 cores. Figure 21 shows the histogram of the time taken for the 39681 cores. Based on these times, it is clear that the compute load is much better balanced in

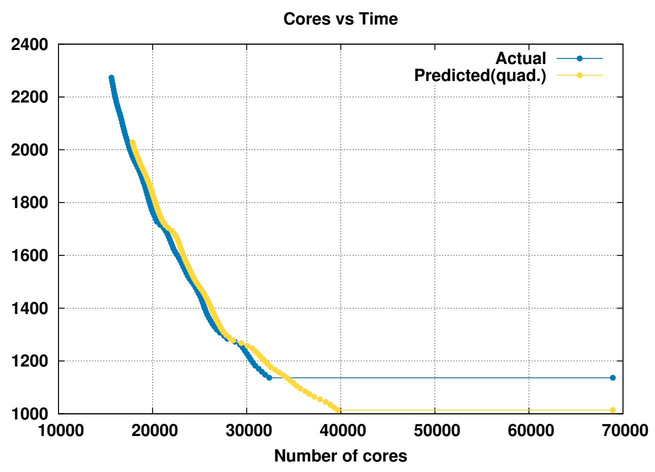

Fig. 19. Number of cores vs time for 68921 parameter combinations using actual and predicted times

this case, resulting in a better parallel efficiency of 89%. The other two mappings results in a higher times.

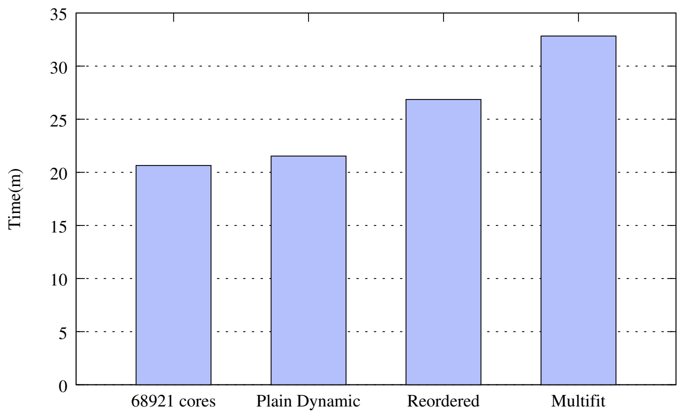

Fig. 20. 68921 problem with 39681 processes using predicted times with different mappings

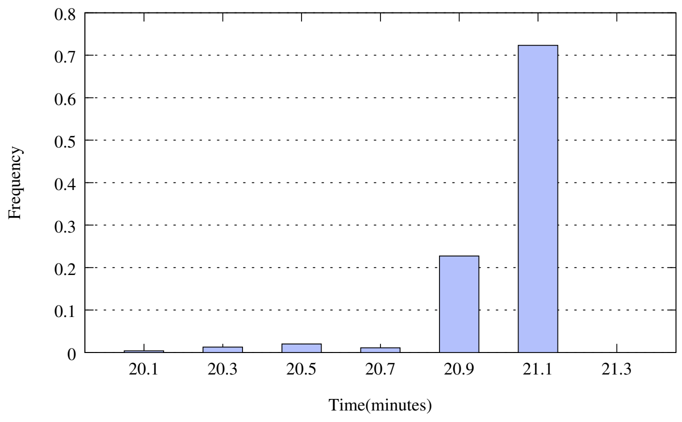

Fig. 21. Histogram of time taken for 68921 parameter combinations using 39681 cores (Dynamic Load balance)

We next evaluate these load balancing strategies when we know the actual time taken for all the 68921 parameter combinations using 68921 cores. The purpose of this exercise is to determine whether better time prediction gives scope for further improvement in load balancing. Figure 22 shows the timings with 32405 cores with plain dynamic load balancing, dynamic load balancing with the task list sorted based on times, and the multifit based static mapping. The multifit based static mapping and the reordered load balancing perform close to the optimum that uses 68921 cores. This results in achieving a parallel efficiency of 94.6%. Using fewer resources, the simulation results are obtained with roughly the same time as with using 68921 cores. This is a significant improvement compared to the parallel efficiency of 51% when 68921 cores are used. Figure 23 shows the histogram of the time taken for the 32405 cores which indicates better load balancing across the cores.

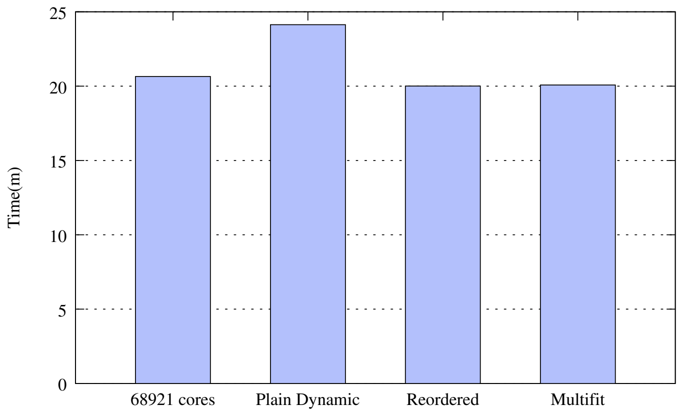

Fig. 22. 68921 problem with 32405 processes using actual times with different mappings

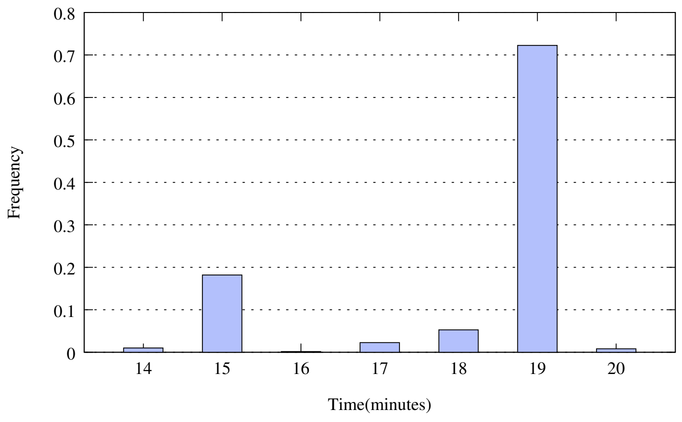

Fig. 23. Histogram of time taken for 68921 parameter combinations using 32405 cores (STATIC mapping)

An even higher parallel efficiency of 98.2% can achieved using just 25810 cores with a 14.5% increase in the time taken. As shown in figure 24, the multi-fit based static mapping performs best. (We chose to evaluate this result on 25810 cores because bin-packing results indicated this as the largest core count for which one could exceed 98% parallel efficiency.)

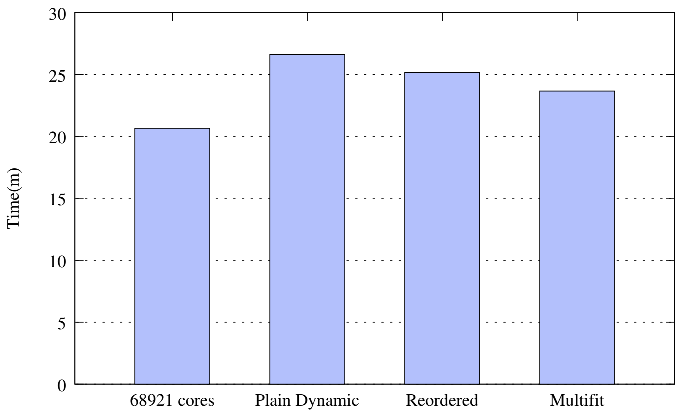

Fig. 24. 68921 problem with 25810 processes using actual times with different mappings

The above methods assume the availability of timings from smaller runs, based on which we predict the optimal number of cores and the mapping to use for a much larger problem size. We now discuss a rule of the thumb when such data is not available. We can use the List Scheduling based dynamic load balancing algorithm with half the number of cores as the number of parameters. Table II shows timings for three different problem sizes using this rule of the thumb. It shows that we can reduce the number of cores by half, at the expense of 20% to 25% increase in the overall time.

Number of parameters	Number of cores	Plain Dynamic	Optimal
1331	666	20.48s	16.91s
9261	4361	23.58s	18.81s
68921	34460	22.78s	18.93s

TABLE II
List Schedule based dynamic load balancing with half the number of cores as the number of parameters.

VII. Conclusions and future work

We have identified some major bottlenecks to parallelization of the SPED model. One of the bottlenecks was related to the workflow, which had a sequence of components. Including some aspects of the validation into the basic simulation, apart from a few simple optimizations in the sequential code, reduced the time substantially. Parallel read of a common input file proved to be another bottleneck which was resolved. Finally we showed that our time prediction scheme can be used to estimate a much smaller number of cores on which a simple dynamic load balancing scheme can yield result close to optimum.

The SPED model has important applications to public health, and so this work is of much relevance. Our preliminary results on the application science has suggested that certain alternate deplaning strategies could reduce the number of human-human and human-surface contacts substantially. In addition, our results show that if vacant seats are available, then assigning them to the middle section of the plane is preferable, and assigning them to middle seats in a row of three seats is preferable.[4]

One aspect of our future work is guided by our observation that with more accurate time prediction, the number of cores needed could be reduced further without increasing the total computational time. Improving time prediction will be an important component of future work. On the other hand, empirical results also suggest that, in the absence of timing results from preliminary runs, the simple dynamic load balancing with half the number of cores as the number of parameters, would perform reasonably well. Yet another scope for optimization is reducing the parameter space swept. In order to generate different scenarios, it may be adequate to sample regions of parameter space more coarsely if results are not sensitive to parameters in that region, while sensitive regions ought to be sample on a finer scale.

Acknowledgment

This material is based upon work supported by the National Science Foundation ACI under grants #1525061, #1524972, and #1525012 (Simulation-Based Policy Analysis to Reduce Ebola Transmission Risk in Air Travel). Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. We thank NCSA for providing access to the Blue Waters supercomputer. This research is performed as a part of Project VIPRA (Viral Infection PRopagation through Air-travel). We acknowledge useful discussions with VIPRA team members Mathew Scotch and Robert Pahle. We acknowledge IBM Research for providing us access to the BlueGene/Q and XSEDE for providing access to Stampede.

References

[1] *VIPRA (Viral Infection Propagation Through Air-Travel) - Science Based Policy Analysis:* http://www.cs.fsu.edu/vipra/

[2] *Centers for Disease Control and Prevention. Epidemiological notes and reports. Interstate importation of measles following transmission in an airport California, Washington*. MMWR-Morbidity and Mortality Weekly report, 32 210-216, 1982.

[3] M. Isacson and J. A. Frean, *African malaria vectors in European aircraft.* The Lancet, 17 Vol. 357, No.9251, 2001, pp. 235.

[4] M. R. Moser, T. R. Bender, H. S. Margolis, G. R. Noble, A. P. Kendal and D. G. Ritter. *An outbreak of influenza aboard a commercial airliner.* 1979. American Journal of Epidemiology, 110(1), 1-6.

[5] S. J. Olsen, H. L. Chang, T. Y. Y. Cheung, A. F. Y. Tang, T. L. Fisk, S. P. L. Ooi, and J. Lando, *Transmission of the severe acute respiratory syndrome on aircraft.* New England Journal of Medicine, Vol. 349, No.25, 2003, pp. 24162422.

[6] M. Tracy, *Transmission of tuberculosis during a long airplane flight.* N Engl. J Med, 335, 675, 1996.

[7] M. A. Widdowson, R. Glass, S. Monroe, R. S. Beard, J. W. Bateman, P. Lurie, and C. Johnson. *Probable transmission of norovirus on an airplane.* Jama, 293(15), 1855-1860, 2005.

[8] S. Namilae, *Simulation of passenger evacuation form airplane using molecular dynamics simulations*, Boeing Technical Excellence Conference, St Louis, 2014.

[4]These results suggest strategies for the entire set of passengers. They do not indicate that a particular passenger will benefit from avoiding the middle section of the plane or the middle seat.

[9] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. *Understanding and improving computational science storage access through continuous characterization.* In ACM Transactions on Storage, 7:8:1-8:26, October 2011.

[10] Konstantinos Chasapis, Manuel F. Dolz, Michael Kuhn and Thomas Ludwig *Evaluating Lustre's Metadata Server on a Multi-Socket Platform*, 9th Parallel Data Storage Workshop, in conjunction with SC14, 2014, New Orleans, LA.

[11] Torben Kling Petersen, *Inside The Lustre File System*, SEAGATE Technology paper.

[12] R.L. Graham, E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan, *Optimization and approximation in deterministic sequencing and scheduling: a survey*, Ann. Discrete Math. 5, 287-326 (1979).

[13] E.G. Coffman Jr., M.R. Garey and D.S. Johnson, *An application of bin-packing to multiprocessor scheduling.* SIAM Journal on Computing, Vol. 7, 1978, 117.

[14] Dorit S. Hochbaum and David B. Shmoys, *Using Dual Approximation Algorithms for Scheduling Problems: Theoretical and Practical Results.* Journal of the Association for Computing Machinery. Vol. 34. No. I. January 1987

[15] M. Garey and R. Graham, D. Johnson and A. Yao, *Resource constrained scheduling as generalized bin packing,* Journal of Combinatorial Theory Ser. A, 21 (1976) 257-298

[16] D.S. Johnson, M.R. Garey, *A 71/60 theorem for bin-packing.* Journal of Complexity 1, 65-106 (1985)

[17] W. Fernandez de la Vega and G. S. Lueker, *Bin packing can be solved within 1 + ϵ in linear time*, Combinatorica, 1 (1981), pp. 349—355.