Scalability, Locality, Partitioning and Synchronization in PDES*

David M. Nicol
Department of Computer Science
Dartmouth College
Hanover, NH 03755
nicol@cs.dartmouth.edu

1 Introduction

Scalability analysis asks how performance of a certain application (or application class) behaves as the application problem size increases and the parallel architecture executing it increases. Scalability studies generally concern very specifically defined applications, e.g. , FFT, global reductions, and many other precisely defined operations [4].

Parallel and distributed discrete-event simulation (PDES) is a critical technology for an important class of very large complicated simulation models. However, with few exceptions, the bulk of empirical work in PDES has been on relatively small models. Furthermore, synchronization behavior is frequently complicated, which makes it very difficult to analytically prove anything about performance executing large models on large machines.

Since the behavior of a PDES system depends very much on the model being simulated, one cannot expect a scalability analysis to be as precise as that of a very specific algorithm. What we *can* do is look for general characteristics of the simulation model and the partitioning algorithm that yield scalable behavior. That identification is one main point of this paper. The scalability analysis we do is quite simple. The value lies in identifying general problem and partition characteristics which, when taken as starting assumptions, yield that straightforward analysis. Our other main point is to demonstrate circumstances under which a simple conservative synchronization protocol is scalable, and to show how to address an inherent tradeoff between synchronization overhead and load imbalance.

In [6] Lubachevsky argues for the scalability of the Bounded Lag synchronization protocol. In contrast to our analysis here, his involves a more specific model of simulation and workload behavior, and does not explicitly consider the communication overheads as we do (and as we will see, the communication overheads are the factor most limiting scalability). In [7] it is show that the YAWNS protocol keeps overheads due to (quote) "synchronization, processor idle time, lookahead calculation, and event list manipulation" to within a constant factor of optimal. Our model resembles this in some aspects, but with a focus on communication, and identification of the mitigating effects of growing the simulation model faster than the architecture.

2 Model

Inspired by VHDL [1] we think of the simulation model as a network of *entities*, connected by *channels*. Entities communicate exclusively by sending messages over channels. A channel may model a communication link in the simulation model, or may be a logical connection that has no physical reflection in the system being simulated. In the latter case there is usually no simulation delay attributed to the communication, in the former case there is. When a delay is associated with the action of sending a message, we have *lookahead*. That is to say that when the message is initiated by the sender, one can infer that the message will not affect its recipient at least for as long as the (simulated) time it takes for the message to reach the recipient.

Without loss of generality we assume that all channels have non-zero delays, and are uni-directional. Aggregation can transform a model with zero delays into one without, at the cost of some parallelism. We assume that every channel delay is at least $\delta > 0$ large. This is a natural assumption for simulation models of physical systems.

For every entity i we let λ_i be its average *event rate*, in units of events per unit simulation time. λ_i quantifies the average number of *all* simulation events associated with the entity per unit simulation time. For every channel k we let μ_k be the average number

*This work was supported in part by NSF grant CCR-9625894 and DARPA contract N66001-96-C-8530.

of messages sent over the channel per unit simulation time.

A partition $\mathcal{P}$ is a mapping from the set of all entities into the set of all processors (assumed to be homogeneous in processing power). For a given partition $\mathcal{P}$ we let $\Lambda_j(\mathcal{P})$ be the sum of event rates of entities assigned to processor j. We assume that the cost of accessing the event list is proportional to the logarithm of the number of events in the list, which we assume is proportional to the aggregate event-rate on the host processor. The first assumption follows from the use of a standard event list management data structure; the second results from the observation that a posted event exists in the event-list from its send-time to its receive-time. The constant of proportionality depends on the average length of simulation time an event remains in the list. Consequently we model the cost of executing an event on processor i under partition $\mathcal{P}$ as $X \log \Lambda_j(\mathcal{P})$, where X is the problem-dependent constant of proportionality.

A partition induces communication between processors. The rate of communication (in messages per unit simulation time) from processor i to j, denoted $c_{i,j}(\mathcal{P})$, is determined by summing the rates μ_k of channels from entities assigned to processor i to entities assigned to processor j. We assume that the cost of sending messages from one processor to another is no greater than some constant of proportionality M times the number of messages sent, times some function $a()$ of the architecture size. We allow contention in the communication network (its cost is hidden in M), and assume that architecturally motivated growth in contention is captured by $a()$. We model message receipt costs somewhat differently, as messages are converted into events, and hence there is a cost associated with inserting the new event into the event list. The per-message receipt cost is taken to be some constant R times the logarithm of the size of the event list. Finally, we define $\delta(\mathcal{P})$ to be some "natural" interval of simulation time during which we will assess costs. The dependence on partition $\mathcal{P}$ (and tacitly, the number of processors) is included for generality; different synchronization schemes may give rise to different notions of what is "natural".

With these definitions and assumptions in hand, we model the amount of time (excluding synchronization overhead) required by processor i on a machine of N processors, to simulate a epoch of $\delta(\mathcal{P})$ simulation time units by

$$\begin{aligned} T_i(\mathcal{P}) &\leq \delta(\mathcal{P})(X\Lambda_i(\mathcal{P}) \cdot \log \Lambda_i(\mathcal{P}) \\ &\quad + \sum_{j \neq i} (a(N) \cdot M \cdot c_{i,j}(\mathcal{P}) \\ &\quad + R \cdot \log \Lambda_i(\mathcal{P}) \cdot c_{j,i}(\mathcal{P}))) \qquad (1) \\ &= \delta(\mathcal{P})\mathcal{R}_i(\mathcal{P}), \qquad (2) \end{aligned}$$

where $\mathcal{R}_i(\mathcal{P})$ denotes the weighted sum of event rates. The "bottleneck" value is the greatest epoch execution time : $T_{\max}(\mathcal{P}) = \max_i\{T_i(\mathcal{P})\}$. The bottleneck rate sum is

$$\mathcal{R}_{\max}(\mathcal{P}) = T_{\max}(\mathcal{P})/\delta(\mathcal{P}),$$

and clearly is the rate sum associated with the processor defining the bottleneck value.

We assume that all processors execute their work concurrently, and that any degradation in communication concurrency is already captured by constant M and function $a()$.

The formulation so far has neglected the extra overhead induced by synchronization. We model this with an "overhead intensity function" $S(\mathcal{P})$ such that the synchronization overhead (including blocking) suffered by a processor in an epoch of $\delta(\mathcal{P})$ time is the product $S(\mathcal{P}) \cdot \delta(\mathcal{P})$. For most synchronization protocols, quantifying $S(\mathcal{P})$ is extremely difficult. While we can and will instantiate it for a simple protocol, our purpose now is to identify that overhead so we can reason about it when we consider scalability issues. The total cost of executing $\delta(\mathcal{P})$ units of simulation time is taken to be no greater than $T_{\max}(\mathcal{P}) + S(\mathcal{P}) \cdot \delta(\mathcal{P})$. This measure ignores stochastic effects that variation in inter-event time distributions may create. Those effects are second-order as compared to effects of load imbalance and communication overload, and so in the interests of exposition are not treated here.

A usual measure of assessing parallel performance is the *processor utilization*, defined as the fraction of time a processor spends doing work that is done in an equivalent serial simulation. Letting $\Lambda = \sum_i \lambda_i$ be the sum of event rates in all entities, the utilization is bounded by

$$\begin{aligned} U(\mathcal{P}) &\geq \frac{\delta(\mathcal{P}) \cdot \Lambda \cdot X \log \Lambda}{N \cdot (T_{\max}(\mathcal{P}) + S(\mathcal{P}) \cdot \delta(\mathcal{P}))} \\ &= \frac{\Lambda \cdot X \log \Lambda}{N \cdot (R_{\max}(\mathcal{P}) + S(\mathcal{P}))}. \qquad (3) \end{aligned}$$

This ratio is obtained by comparing the total execution time during one epoch of simulation in a serial simulation to the aggregate execution time of all N processors in a parallel simulation in that same epoch. Speedup can be computed by multiplying utilization times the number of processors. If we can show that utilization remains bounded from below as the problem size and architecture simultaneously grow, then we will have demonstrated scalability.

3 Scalability

Now we consider how this lower bound on utilization behaves as the simulation model and architecture size change. First we identify constraints on how the simulation model grows, that yields by example the possibility of a partitioner finding a partition that perfectly balances workload, to within a constant factor. With this result we are able to rewrite the lower bound on utilization in a way that gathers terms relating to load imbalance, communication overhead, and synchronization overhead. Next we consider the effect of increasing the model size or complexity without changing the architecture. We note that even in this most benign form of growth, the possibility exists for decreasing utilization due to growth in the synchronization term. Next we consider simultaneous growth in problem size and architecture, a consideration that reveals the necessity of locality in communication if scalability is to be achieved.

3.1 Maintaining Workload Balance

The first problem we face is characterizing increasing simulation model size. Without a specific parametric description of the simulation model, we can only identify constraints we place on how a simulation model might grow. We let "entity growth function" $g(N)$ give the number of entities as a function of the number of processors N; for notational simplicity we assume that N divides $g(N)$ evenly. We constrain the growth of the model by three assumptions:

Workload Growth Assumptions

- for every entity i, the event rate $\lambda_i = O(\frac{\Lambda^{(N)}}{N})$, where $\Lambda^{(N)} = \sum_{i=0}^{g(N)-1} \lambda_i$,
- for every entity i, the sum of event rates on all outgoing channels is $O(\lambda_i)$,
- every message received by an entity generates an "event" for the entity.

The first constraint means that the model does not grow in such a way that one entity becomes a serial bottleneck. If we are to balance workload it is the weakest possible constraint, in that it permits one entity to have a workload so great one might place only it on a processor. The second constraint links possible communication rates with event processing rates in a natural way; the use of the big-oh notation is precise—it gives an asymptotic upper bound and says nothing at all about any lower bound. The third constraint implies that the earlier bound on event rates also bounds incoming communication volume. Under these conditions, as the model and architecture grow, we can always bound the growth of $R_{\max}(\mathcal{P})$ from above. Showing this requires some work.

We first show that under the constraints outlined above, we can find a partition that balances workload to within a constant factor of optimal, which is to say that the largest event rate on any processor is $O(\Lambda^{(N)}/N)$. To demonstrate this result we look at one simple minded way of partitioning and show that it achieves this bound. Better partitioners can do at least as well. Given entity event rates (say, by observation), we renumber them by order : $\lambda_0 \geq \lambda_1 \geq \cdots \lambda_{g(N)-1}$. Consider the partition $\mathcal{P}'$ that maps entity i to processor $i \bmod N$. It is not difficult to see that the sum of event rates on processor 0 is maximal. We now consider how to make that sum as large as possible if we view the event rates as variable, subject only to the constraint that the relative orderings remain the same.

Fix λ_0. Note that any assignment of ordered event rates that maximizes the load given to processor 0 has $\lambda_1 = \lambda_2 = \cdots = \lambda_N$, for suppose not. If $\lambda_j > \lambda_N$ and $1 \leq j < N$, we can reassign the arithmetic average of rates $\lambda_j, \lambda_{j+1}, \ldots, \lambda_N$ to entities j through N, thereby raising the rate assigned to entity N, which is assigned to processor 0. This contradicts the assumption that the event load assigned to processor 0 is as high as possible. The same argument applies to entities with indices between $N+1$ and $2N$: they are the same as λ_{2N}, otherwise an averaging process can raise λ_{2N} and hence raise the total rate assigned to processor 0. Continuing on in this vein we establish that if the entities assigned to processor 0 have rates $\lambda_0, \lambda_N, \lambda_{2N}, \ldots, \lambda_{g(N)-N}$, then under a rate assignment that maximizes processor 0's load we have

$$\Lambda^{(N)} \geq \lambda_0 + (N-1) \times (\lambda_N + \lambda_{2N} + \cdots + \lambda_{g(N)-N}).$$

λ_0 is a lower bound on processor 0's load, the product involving $N-1$ sums a common lower bound on the load of all other processors (the bound is not an equality because the bound omits one rate—the least one for each of the last $N-1$ processors). With straightforward algebraic manipulation we establish an upper bound on the event rate of the most heavily loaded processor:

$$\Lambda'_0 \leq \lambda_0 + \frac{\Lambda^{(N)} - \lambda_0}{N-1}.$$

Viewed as a function of λ_0, we note that the right-hand-side is increasing in λ_0 and so is maximized when λ_0 is as large as it can be. By assumption there is a constant c_0 such that $\lambda_0 \leq c_0 \Lambda^{(N)}/N$ for all N,

whence

$$\begin{aligned}\Lambda'_0 &\leq c_0\frac{\Lambda^{(N)}}{N} - c_0\frac{\Lambda^{(N)}}{N(N-1)} + \frac{\Lambda^{(N)}}{N-1} \\ &= c_0\frac{\Lambda^{(N)}}{N} - c_0\frac{\Lambda^{(N)}}{N(N-1)} + \frac{\Lambda^{(N)}}{N} + \frac{\Lambda^{(N)}}{N(N-1)} \\ &\leq (c_0+1)\frac{\Lambda^{(N)}}{N} \qquad (4)\end{aligned}$$

where in the last step we take $c_0 \geq 1$, which we can do without loss of generality. Thus we have proven that the sum of event rates on the most-heavily loaded processor is $O(\Lambda^{(N)}/N)$.

If the event rate on each processor is bounded from above, then our second and third assumptions allow us to bound the cost of sending and receiving messages. For if $c_1 = c_0+1$ and $\Lambda'_i \leq c_1\Lambda^{(N)}/N$ for all sufficiently large N, there are constants c_2 and c_3 such that

$$\begin{aligned}&a(N)M\sum_{i\neq j} c_{i,j}(\mathcal{P}') + R\cdot\log\Lambda'_i\sum_{i\neq j} c_{j,i}(\mathcal{P}') \\ &\leq (a(N)M\cdot c_2 + R\cdot c_3\log(c_1\frac{\Lambda^{(N)}}{N}))\times\frac{c_1\Lambda^{(N)}}{N},\end{aligned}$$

for all sufficiently large N. This then demonstrates that

$$\begin{aligned}R_{\max}(\mathcal{P}') &\leq X\cdot c_1\frac{\Lambda^{(N)}}{N}\log(c_1\frac{\Lambda^{(N)}}{N}) \\ &+ (a(N)M\cdot c_2 + R\cdot c_3\log(c_1\frac{\Lambda^{(N)}}{N}))\times\frac{c_1\Lambda^{(N)}}{N} \qquad (5)\end{aligned}$$

for all sufficiently large N.

Substituting this bound on $R_{\max}(\mathcal{P}')$ into the bound given in (3) and performing some rearrangement gives the expression as a lower bound on $U(\mathcal{P}')$. Later in the text we refer to this as expression (6).

$$\begin{aligned}&\frac{X\log\Lambda^{(N)}}{c_1X\log(\frac{c_1\Lambda^{(N)}}{N}) + a(N)Mc_1c_2 + Rc_3c_1\log(\frac{c_1\Lambda^{(N)}}{N}) + \frac{S(\mathcal{P}')}{\Lambda^{(N)}/N}} \\ &= \frac{X}{c_1(X+Rc_3)(1+\log(\frac{c_1}{N})) + \frac{a(N)Mc_2c_1}{\log\Lambda^{(N)}} + \frac{N}{\Lambda^{(N)}}\frac{S(\mathcal{P}')}{\log\Lambda^{(N)}}}.\end{aligned}$$

When parsed, the LHS of expression (6) intuitively relates the cost of executing an event in a serial simulation to the amortized cost of executing that event in a parallel simulation. The denominator reflects the smaller cost of executing the event in parallel, reflects bounded per-event average communication overhead and bounded per-message average overhead due to a message receipt, and gives the average per-event synchronization cost. The RHS collects terms for later analysis.

A precise definition of scalability is more technical than it is informative. However, if we are interested in scalability then we are interested in how performance "projects" as the model problem grows and/or as the simulation engine used grows. There are a number of different scenarios, which we now consider.

3.2 Increasing Model Complexity

One way a model "grows" is if the events become more complex, if they involve more computation because the model they describe has increasingly more detail. If that complexity does not engender more communication or synchronization overhead, in expression (6) the increase in model complexity manifests itself as an increase in the magnitude X. All other things being equal, this increases the value of $U(\mathcal{P}')$, so a change of this type "scales".

The result above rests on a strong assumption though, that the increasing model complexity does not affect the synchronization overhead. It well may. In an optimistic protocol the state-saving cost may rise and/or the risk of rollback increase. In a conservative protocol more model complexity generally implies smaller lookahead, which would also serve to increase the synchronization overhead. In this more complicated scenario, the bound on $U(\mathcal{P}')$ can increase or decrease, depending on how quickly $S(\mathcal{P}')$ changes with increases in X. Given a mathematical form for $S(\mathcal{P}')$ one could express the condition formally using derivatives. For now we just observe the potential for linkage between model complexity and synchronization overhead.

Another way a model might grow is if the event rates increase, without affecting X. This is a fairly typical mode of model growth: given a submodel template (e.g. an ATM switch), one "grows" the model by replicating the submodel and inter-connecting the replications. This translates into an increase in $\Lambda^{(N)}$, which serves to increase the bound on $U(\mathcal{P}')$. However, in principle the synchronization cost per event might again rise. For instance, in an optimistic protocol the threat of rollback might be larger—especially if the increasing number of events exacerbates a less-than-perfectly balanced workload.

What we can conclude is that if the architecture remains constant and the simulation workload is increased, the utilization *may* rise, depending on how the synchronization overhead behaves as the workload increases. If that overhead does not increase or does not increase too fast, then the simulation system "scales".

3.3 Simultaneous Growth in Model and Architecture

Now we consider how functional forms for various factors in expression (6) must relate to each other if

the simulation is to scale. We do so noting that physical machine limits exist. In practice this means that subject to upper bounds on N and problem size, we consider how problem size must grow with respect to N if the bound in expression (6) is to be kept from degrading. In particular, we consider how the ratios $a(N)/\log\Lambda^{(N)}$ and $(N/\Lambda^{(N)})(S(\mathcal{P}')/(\Lambda^{(N)}))$ are to behave as N grows.

Consider architectural cost $a(N)$ first. Recalling that $a(N)$ reflects the average path length of message communication, whether or not communication delays grow in N depends very much on how elements of the model communicate. Local communication patterns that are sustained as the model grows and are exploited by the partitioner, keeps $a(N)$ constant. However, if on average communication must cross the entire parallel architecture, then $a(N)$ grows in proportion to network diameter. For a communication architecture with an underlying hypercube this would be $a(N) \propto \log N$; for a mesh or torus in d dimensions we have $a(N) \propto N^{1/d}$. In the former case, if $\Lambda^{(N)}$ grows in proportion to N (as it will if the average workload per processor is kept constant as the architecture grows) then the ratio $a(N)/\log\Lambda^{(N)}$ increases in N but approaches a fixed upper bound, which means in our asymptotic analysis we can ignore the communication term. In the general case, in order to keep the communication term from growing, $\Lambda^{(N)}$ must grow faster than does N, specifically, $a(N) = O(\log\Lambda^{(N)})$. While we can increase problem size faster than architecture size, ultimately we exhaust memory. Whether one does this before exhausting the supply of available processors is a matter of constants of proportionality. The key point is that if $a(N)$ grows faster than $\log N$, to maintain a consistent level of performance, the problem size must grow faster than the architecture. The relationship $a(N) = O(\log\Lambda^{(N)})$ describes precisely the needed rate of growth.

Now consider synchronization costs. If $\Lambda^{(N)}$ grows in proportion to N, then to keep the synchronization term in expression (6) from growing it is necessary that $S(\mathcal{P}')$ not grow any faster than $\log N$. Alternatively, if $S(\mathcal{P}')$ grows faster than $\log N$, then $\Lambda^{(N)}$ must grow fast enough so that $\Lambda^{(N)}\log\Lambda^{(N)}$ keeps pace with $N \cdot S(\mathcal{P}')$. Without a functional form for $S(\mathcal{P}')$ it is impossible to assert whether this condition is met or not. However, for an important class of synchronization algorithms we *can* ascribe form to $S(\mathcal{P}')$, for all conservative window-based algorithms (e.g. [7, 2, 5]) rely in the end on barrier synchronizations or global reductions, which can usually be implemented with cost proportional to the diameter of the architecture. In the case of a hypercube-based design, $S(\mathcal{P}') \propto \log N$, and the synchronization term is bounded.

4 Scalability and Optimization of QS

Next we add some concreteness to this discussion by looking at a specific synchronization protocol we call *Quanta Synchronization*, or QS. First we show that the method can scale, depending on the assumed workload growth. Second, we consider how to deal with an inherent tradeoff between load imbalance and synchronization overhead.

4.1 QS Scales

Our workload model lends itself to coordination through a very simple synchronization mechanism. Recall that every channel is assumed to have a delay that is bounded sharply from below by some δ. If we synchronize all processors every δ quanta of simulation time, we are assured that no event occurring in one processor can affect the state of another processor in that same time quanta. QS is the protocol used to synchronize the Wisconson Window Tunnel[9]. A variant of it synchronizes **Nops** [8].

For QS we can quantify the per-unit simulation overhead term $S(\mathcal{P}')$ from expression (6): $S(\mathcal{P}') \propto r(N)/\delta'(\mathcal{P}')$, where $r(N)$ is the cost of implementing a min-reduction on N processors. $r(N)$ depends on the topology of the interconnection network, in the same way that function $a(N)$ does, discussed earlier. Here we explicitly choose the time-quanta to be the minimum channel delay among all channels cut by the partition.

In order to keep the synchronization term of expression (6) from growing, we require that its denominator dominate its numerator, which is to say there is a constant c_4 such that for all N

$$N \cdot r(N) \leq c_4 \cdot \Lambda^{(N)} \cdot \delta(\mathcal{P}') \cdot \log\Lambda^{(N)}.$$

Rewriting, we require that

$$\begin{aligned} r(N) &\leq c_4 \cdot \frac{\Lambda^{(N)}}{N} \cdot \delta(\mathcal{P}') \cdot \log\Lambda^{(N)} \\ &= O\left(\frac{\Lambda^{(N)}}{N} \cdot \log\Lambda^{(N)}\right). \end{aligned}$$

If the underlying network has logarithmic diameter, then $r(N)$ grows no faster than $\log N$, and so long as $\Lambda^{(N)}$ grows linearly in N this relationship is satisfied. If $r(N)$ grows faster than $\log N$, the relationship indicates just how fast $\Lambda^{(N)}$ must grow as a function of N. For instance, if $r(N) \propto N^{1/2}$ (as in a mesh connection architecture) then if average processor workload

$\Lambda^{(N)}/N$ grows as fast as $N^{1/2}/\log\Lambda^{(N)}$, then the synchronization overhead is bounded.

Assuming the "worst case" that $a(N) \propto r(N)$ (locality of communication can reduce $a(N)$), we see that the communication requirement of $a(N) = O(\log\Lambda^{(N)})$ is more stringent than the synchronization requirement above. If communication scales, then so does QS synchronization.

Pulling all of this together, we've demonstrated that a simulation model that obeys the Workload Growth Assumptions detailed in subsection 3.1, synchronized using QS, can be scalable so long as the growth of simulation model as a function of architecture size is sufficient to dampen both the communication overhead and the synchronization overhead. The needed growth is never more than $\log\Lambda^{(N)} \propto a(N)$, and may be considerably less.

4.2 The Synchronization / Load Imbalance Tradeoff

In the problem class under consideration, QS exploits lookahead obtained from minimum channel delays on channels that are "cut" by the partition. The larger the minimum such channel delay, the less frequently the processors must synchronize. This creates a trade-off between synchronization overhead and load imbalance. The larger we insist that the minimum channel delay be on a cut channel, the fewer options the partitioner has, because entities connected by channels with smaller delays must be co-resident. The fewer options it has, the worse the load imbalance will be. We point out here how a partitioning strategy can address this tradeoff efficiently.

Imagine that we insist on having the minimum cut channel delay be at least d, and will use d as the synchronization quanta. Given a partitioning algorithm, we can find a partitioning $\mathcal{P}_d$ by first collapsing the network of entities, merging any two entities with a minimum channel delay between them less than d. Applied transitively, one creates a smaller network, but one in which all exposed channels have minimum delay at least as large as d. The partitioner can then be applied to that reduced network.

We assume, not unreasonably, that the larger d is, the worse load imbalance under $\mathcal{P}_d$ will be. Using the notation of section 2, this means that $R_{\max}(\mathcal{P}_d)$ (the execution time per unit simulation time) increases in d. However, the synchronization cost per unit simulation time goes down as d goes up. Ultimately we seek the partition that minimizes the overall performance rate $R_{\max}(\mathcal{P}_d) + r(N)/d$, where $r(N)$ is the delay cost of implementing the global synchronization.

For a given partitioning algorithm, we can find the partition that addresses this tradeoff nearly optimally, provided we can assume that if $d < d'$, then $R_{\max}(\mathcal{P}_d) \leq R_{\max}(\mathcal{P}_{d'})$. Our notion of "best" will be approximate, in the following sense. We will discretize the range of potential values of $R_{\max}(\mathcal{P}_d)$ and solve an optimization problem, finding a solution that is optimal up to the granularity of that discretization. Given that we are interested in balancing load imbalance against synchronization overhead, and the synchronization cost $r(N)$ is known, we will choose a discretization unit that is some fixed fraction of $r(N)$, e.g., $\Delta = r(N)/F$, for some F.

Our strategy is to craft a "probe" function that, for a given level of performance, determines whether it is possible to partition the network and achieve that performance. We then search a space of discretized performance levels using binary search, at each level applying the probe function to determine feasibility.

To motivate the searching approach consider the following. If we apply the partitioning algorithm without any constraints whatsoever on the channels, we will find the partition that best balances the workload ("best" subject to the use of the particular partitioning algorithm, that is). The execution time per unit simulation time of that partition ($\mathcal{P}_0$) is comprised of two parts, the execution time for event processing (and communication) and the execution time for synchronization: $R_{\max}(\mathcal{P}_0) + r(N)/\delta'(\mathcal{P}_0)$. To express the event processing component as a multiple of the performance level discretization we compute $n_0 = \lceil R_{\max}(\mathcal{P}_0)/\Delta \rceil$, so that $R_{\max}(\mathcal{P}_0) \approx n_0\Delta$. This expression is a lower bound on the optimal cost per unit simulation time (because $R_{\max}(\mathcal{P}_0)$ is minimal); the full cost of partition $\mathcal{P}_0$ is an upper bound on the optimal achievable performance. We thus have a closed range of performance values, discretized to resolution Δ. We denote the number of such levels by

$$\begin{aligned} L(\Delta) &= \lceil \frac{R_{\max}(\mathcal{P}_0) + r(N)/\delta(\mathcal{P}_0) - R_{\max}(\mathcal{P}_0)}{\Delta} \rceil + 1 \\ &= \lceil \frac{r(N)/\delta(\mathcal{P}_0)}{\Delta} \rceil + 1 \\ &= \lceil \frac{r(N)/\delta(\mathcal{P}_0)}{r(N)/F} \rceil + 1 \\ &= \lceil \frac{F}{\delta(\mathcal{P}_0)} \rceil + 1. \end{aligned} \tag{6}$$

Now imagine the existence of a probe function that takes a target performance level and determines whether it is possible for the partitioning to achieve that level. We can do a binary search over the discretized space and so determine the lowest achievable cost, up to the resolution of the discretization.

We now consider how to construct a probe function. Let $n\Delta$ be in the range of performance levels. One possible way to achieve this level would be if the partitioner was able to achieve execution cost $R_{\max}(\mathcal{P}_d) = R_{\max}(\mathcal{P}_0)$ for d large enough to drive the amortized synchronization cost down to the level needed. We consequently solve for d_1 in equation

$$n\Delta = n_0\Delta + r(N)/d_1,$$

giving $d_1 = r(N)/((n - n_0)\Delta)$. Since in practice $R_{\max}(\mathcal{P}_d)$ may increase as d increases, we know that to achieve the desired level of performance, it is necessary (but not sufficient) to restrict the minimum channel delay to be as large as d_1. Given d_1 we contract the simulation model to eliminate channels with minimum delays smaller than d_1, and apply the partitioning algorithm to the result. This yields a partition $\mathcal{P}_1$ with execution cost per unit simulation time of $R_{\max}(\mathcal{P}_1) + r(N)/d_1$. We discretize the workload level to $n_1\Delta$ by solving for n_1 in equation $n_1 = \lceil R_{\max}(\mathcal{P}_1)/\Delta \rceil$. One of three conditions holds at this point:

$n_1\Delta + r(N)/d_1 = n\Delta$. The probe has determined that the sought performance level can be achieved, and thus returns value **true**.

$n_1 > n$. The sought performance level can never be achieved, because already the cost without synchronization exceeds the target cost, and can only get worse by increasing the minimum channel delay. In this case the probe returns value **false**.

$n_1\Delta + r(N)/d_1 > n\Delta$**.** We cannot yet tell whether the sought level of performance is possible. We *can* tell that if it is possible, then it is only possible by increasing the minimum channel delay. Note that $n_0 < n_1 < n$ in this case.

Should the last condition occur, we repeat the process, first solving for d_2 in the equation

$$n\Delta = n_1\Delta + r(N)/d_2,$$

and using that solution to compute partition $\mathcal{P}_2$, with approximated cost $n_2\Delta + r(N)/d_2$. We run this solution through the same set of three tests. This cycle repeats, every cycle looking to "decay" the synchronization term from the previous iteration's solution to achieve the goal, until the probe returns **true** or **false**. Every step of the process, the coefficient of Δ in the examined solution increases. This shows that the probe must terminate, for eventually the sought performance will be achieved or the coefficient will exceed n.

In any given probe, no more than $L(\Delta)$ iterations of the inner loop will be executed, and there are $\log L(\Delta)$ calls to the probe function. Each iteration of the inner loop of a probe call involves collapsing the network and calling the partitioner. The cost of these activities depends on the partitioner; assuming that the cost of partitioning dominates that of collapsing the network, we see that we can deal with the tradeoff between synchronization overhead and load imbalance at a cost that is $O(L(\Delta)\log L(\Delta))$ times the cost of calling the partitioner once.

Is the extra effect worth the trouble? It all depends. If the unconstrained partition discovers a partition such that the synchronization cost is only 5% of the total cost, then no amount of effort is going to improve performance by more than 5%. If on the other hand the synchronization cost dominates the total cost, then the effort may be well spent. Suppose the synchronization overhead is 10 times that of the execution time, i.e., $R_{\max}(\mathcal{P}_0) = r(N)/(10\delta(\mathcal{P}_0))$. We might choose a workload resolution of $\Delta = r(N)/(10\delta(\mathcal{P}_0))$, to allow the potential for a 10-fold reduction in overall cost. This gives $L(\Delta) = 11$, so the factor $L(\Delta)\log L(\Delta)$ is not so onerous, especially considering that the assumption of making $L(\Delta)$ calls to the partitioner on each probe is quite pessimistic.

5 Event List Costs Revisited

Up until now we have assumed that the cost of executing an event is proportional to the logarithm of the size of the event list. Adherents of the calendar queue method [3] claim an average case *constant* per-event complexity, although in our experience, it is altogether too easy to degrade into linearly increasing performance owing to disadvantageous bin widths. The scalability of the calendar queue mechanism in our context is not clear.

However, in related work we have shown how to exploit the minimal channel delay present in the assumed model, to avoid the calendar queue's shortcomings [8]. The key difference is that we need not do linear insertions once a bin is found. This keeps accesses to the global event list at $O(1)$ on average; the remaining per-event cost increases as a function of an entity's fan-in, essentially a "small" event list, kept ordered, is associated with each entity in each window. This observation forces us to re-examine the present model and analyze its effects on scalability.

Let us assume that the simulation model grows in a way that average entity fan-in does not increase. If a serial simulator is clever enough to exploit the minimum channel delay as does **Nops**, then the LHS of

expression (6) changes by replacing all log terms with constants. The denominator of the RHS has a constant term for execution costs, a term multiplied by $a(N)$ for communication costs, and $(N/\Lambda^{(N)})S(\mathcal{P}')$ as the synchronization cost. The *only* way to keep the communication term from growing without bound is if the simulation model and partitioner can keep $a(N)$ below a fixed upper bound. The only way the synchronization term can be kept from growing without bound is if the workload grows fast enough so that $N/\Lambda^{(N)} \propto S(\mathcal{P}')$. It is also possible to achieve scalability if $a(N)$ grows but the workload grows so much faster than N that the sum of communication and synchronization costs remain bounded.

Reduction of the event-list cost is a sort of good news and bad news deal. It makes a real difference on performance of very large models—but makes scalability more challenging because both the serial and parallel versions can be improved. It remains an open question whether it is "fair" to assume that a serial version would emulate the asymptotically superior parallel approach.

6 Conclusions

As parallel discrete-event simulation becomes increasingly important to the solution of very large systems design problems, it becomes increasingly critical to establish whether PDES technology will scale up with increasing problem size and architecture. In this paper we address the problem in a general setting, provide a resounding conclusion: **maybe.** To scale requires that the simulation model not grow in ways that defeat an ability to load balance, and that do not overwhelm any one processor with communication. It requires an architecture that scales as well. It requires a partitioner that balances workload and exploits locality of communication. The specific partition strategy we examined is very simple, our point is not to promote its specific use. Our point is that scalability is possible using it, and hence if a more refined partitioner can balance workload and maintain locality of communication, then simulations built using it will scale also. If these conditions apply, then we demonstrate by example a simple conservative synchronization protocol, QS, that scales.

Using QS we then examine the trade-off between load-imbalance and synchronization overhead. We show how to efficiently manage that trade-off by probing the space of potential restricted partitions.

References

[1] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann, San Fransico, CA, 1996.

[2] R. Ayani. A parallel simulation scheme based on distances between objects. In *Distributed Simulation 1989*, pages 113–118. SCS Simulation Series, 1989.

[3] R. Brown. Calendar Queues: a fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988.

[4] A. Grama, A. Gupta, and V. Kumar. Isoefficiency function: A scalability metric for parallel algorithms and architectures. *IEEE Parallel and Distributed Technology*, 1(3):12–21, August 1993.

[5] B.D. Lubachevsky. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM*, 32(1):111–123, 1989.

[6] B.D. Lubachevsky. Scalability of the bounded lag distributed discrete-event simulation. In *Distributed Simulation 1989*, pages 100–107. SCS Simulation Series, 1989.

[7] D. Nicol. The cost of conservative synchronization in parallel discrete-event simulations. *Journal of the ACM*, 40(2):304–333, April 1993.

[8] A. Poplawski and D. Nicol. **Nops**: A conservative parallel simulation engine for TeD.

[9] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, pages 48–60, Santa Clara, CA., May 1993.