Clustering Strategies for Cluster Timestamps

Paul A.S. Ward, Tao Huang, and David J. Taylor
Shoshin Distributed Systems Group
University of Waterloo
Waterloo, Ontario N2L 3G1, Canada
{pasward,t6huang,dtaylor}@shoshin.uwaterloo.ca

Abstract

Visualization tools that illustrate communication in parallel programs use Fidge/Mattern timestamps to efficiently answer precedence queries. These timestamps have poor execution efficiency when the number of processes is large, limiting the scalability of the tool. Self-organizing hierarchical cluster timestamps can scale if the clusters they use capture communication locality. However, no clustering algorithm has been presented that enables these timestamps to work. In this paper we evaluate two clustering strategies for such timestamps, one static and one dynamic. The static algorithm was chosen to demonstrate an unproven assumption of cluster timestamps, namely that good clustering will always *yield significant space saving, and to demonstrate that it is possible to select a range of cluster sizes that provide such a savings. We then assessed the merge-on-N^{th}-communication approach. In all but two cases it provides a timestamp size that is with 20% of the best achievable. We present detailed results for the strategies evaluated.*

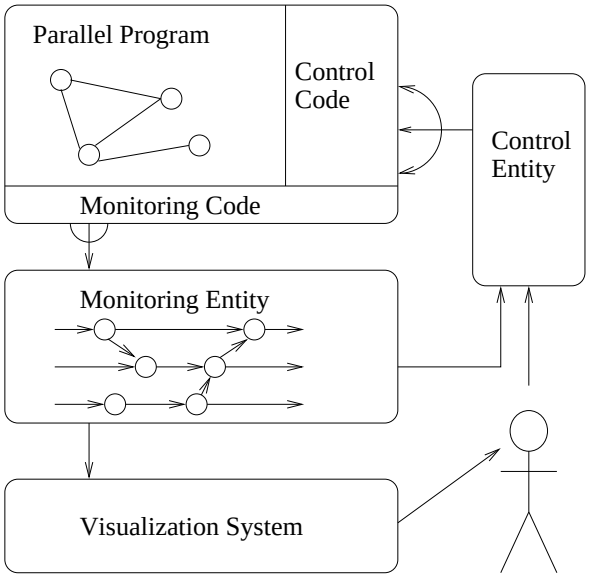

Figure 1. Communication-Visualization Tool Architecture

1 Motivation

Tools that provide communication visualizations for parallel programs, such as ATEMPT [12, 13], Object-Level Trace [8], and POET [14], can be broadly characterized as having the architecture shown in Figure 1. The parallel program is instrumented with monitoring code that captures significant event data. The information collected will include the event's process[1] identifier, number, and type, as well as partner-event identification, if any. This event data is forwarded from each process to a central monitoring entity which, using this information, incrementally builds and maintains a data structure of the partial order of events that form the computation [15]. That data structure may be queried by a variety of systems, the most common being visualization engines and control entities that enable features such as parallel breakpoints (*e.g.* [24]) and program steering (*e.g.* [1]). The efficient representation of this partial-order data structure is the focus of this paper.

Communication-visualization tools we are aware of maintain the transitive reduction of the partial order, typically accessed with a B-tree-like index. This enables the efficient querying of events given a process identifier and event number. It does not, however, enable efficient event-precedence querying, which is one of the most common query types on such structures. To enable such testing, a vector timestamp is computed for each event and stored with that event in the data structure. While a variety of such timestamps have been proposed, the Fidge/Mattern time-

[1] Throughout this paper we will use the term "process" to indicate any sequential entity. It might be a single-threaded process, a thread, a semaphore, an EJB (in the case of Object-Level Trace), a TCP stream, *etc.*

stamp [2, 17] is the one used. Its use is dictated by its ability to answer precedence queries in constant time and the fact that it is dynamic (that is, computable without requiring complete knowledge of the event set). Unfortunately, this timestamp requires a vector of size equal to the number of processes in the computation. Such a requirement does not allow the data structure to scale with the number of processes.

1.1 The Vector-Timestamp-Size Problem

To illustrate this scalability problem consider a thousand-process system, where each process generates a thousand events that are sent to the observation tool. Such a system is by no means extravagant. Indeed there are many systems that already exceed this capacity, and Object-Level Trace has the capacity to monitor such a number of concurrent objects [8]. In such a case, a million events will be generated which will be stored in the partial-order data structure. The Fidge/Mattern vector timestamps for these events must then either be stored with the data structure, or calculated as needed.

If the Fidge/Mattern timestamps are pre-computed and stored with the data structure, we can expect the size of the structure to exceed four gigabytes (a million events times 1000-element vector per process, where each vector element is a 32-bit integer). In such a case the structure will spill into virtual memory. Virtual memory behaves very poorly when dealing with Fidge/Mattern timestamps because their most common use is for precedence testing. Such tests require just a single value from the vector. However, virtual memory systems presume spatial and temporal locality, and thus will read in an entire 4 KB page, or in other words, the complete vector. The rest of the vector typically has no further value. Ward has shown that to do something as simple as computing the greatest-concurrent elements of an event would require about 12,000 pages of virtual memory to be read, only to be discarded [20]. In summary, the virtual-memory system will thrash. The processor cache is similarly of little value with such timestamps.

The alternate solution, and the one adopted by both Object-Level Trace and POET, is to calculate timestamps as required. More precisely, these systems implement their own caching scheme for some timestamps, and calculate forward as needed. The effect is that the precedence-test cost when using such timestamps is $O(N)$, with the size of the constant being a function of the caching approach and the size of the cache (though in all instances it is large). Ward has shown that this approach, while superior to the pre-calculated method, remains quite poor. Elementary operations, such as partial-order scrolling, take several minutes as the vector size approaches 1000 [20], where they take negligible time when the number of processes is small. This remains true even if the number of events is the same in both instances.

The key point that we wish to emphasize here is that the vector-timestamp size is a critical scalability bottleneck. The amount of space consumed by Fidge/Mattern timestamps is sufficiently large as to substantially affect the execution time of systems that use it for precedence determination.

1.2 A Possible Solution

In an attempt to reduce the space-consumption of vector timestamps, Ward and Taylor proposed self-organizing hierarchical cluster timestamps [20, 21, 22]. These timestamps are also dynamic and can efficiently answer precedence queries, but require up to an order-of-magnitude less space than do Fidge/Mattern timestamps. A critical design issue with these cluster timestamps is that the space consumption, and hence efficiency, is heavily dependent on whether or not the clusters used accurately capture locality of communication. If they do not, then the space-consumption saving is substantially impaired. The timestamp algorithm described by Ward and Taylor does not address the issue of cluster selection. Rather, it either assumes that a set of clusters have been pre-determined [21], or it allows for the dynamic clustering of timestamps [20, 22]. In the former case, any clustering strategy can be used, though the only one evaluated was fixed contiguous clusters. In the latter case, the strategy must dynamically cluster processes. In doing so, it can only look at events once, and once a process is placed in a cluster, that placement never changes. Only a trivial clustering algorithm, merge-on-1st-communication, has been evaluated.

The evaluation by Ward and Taylor of their cluster-timestamp algorithm showed that it is possible to substantially reduce the space-consumption using these timestamps. However, they also showed that such a reduction is critically dependent on the cluster size selected. Cluster size is the only parameter that is adjustable for their chosen clustering strategies. Ward subsequently determined that there is no single cluster size that worked well for all computations for the given clustering strategy [20]. The purpose of this paper is to address those two critical problems. Specifically, in this paper we demonstrate three things. First, we show that it is possible to select a single maximum cluster size which produces significant space-savings for all computations we have evaluated to date. Second, we show that for most computations it is possible to select a static clustering algorithm that is not significantly sensitive to the choice of maximum cluster size, and that there exists a large range of maximum cluster sizes which produce near-optimal results for the timestamp. Third, we evaluate some dynamic clustering approaches, and comment on the requirements for such an algorithm.

In the remainder of this paper we first briefly review the operation of the Fidge/Mattern and Ward/Taylor timestamps, and other work that has been developed to address the problem of efficient precedence determination. In Section 3 we describe our clustering algorithm and justify it. We then present experimental results that confirm the value of our cluster-timestamp approach. We conclude by noting what work is still required to make dynamic clustering strategies fully practical.

2 Background

In this section we first briefly describe our parallel computation model. We review the operation of the Fidge/Mattern timestamp. We then describe the self-organizing hierarchical cluster timestamp, taking particular note of how clusters might be selected for these timestamps. Finally, we discuss other approaches to the problem of space consumption, and how they relate to our work.

2.1 Computation Model

We presume a message-passing model of parallel computing: the computation comprises multiple sequential processes communicating *via* message passing. The sequential processes consist of three types of events: send, receive and unary, totally ordered within the process. A *parallel computation* is the partial order formed by the "happened before" relation over the union of all of the events across all of the processes, defined as follows:

Definition 1 (happened before: $\rightarrow \subseteq E \times E$) *"happened before" is the smallest transitive relation satisfying*

1. $e \rightarrow f$ *if* e *and* f *are in the same process and* e *occurs before* f
2. $e \rightarrow f$ *if* e *is a send event and* f *is the matching receive event*

Concurrency between events is then defined as the events not being in the "happened before" relation:

$$e \parallel f \iff e \not\rightarrow f \wedge f \not\rightarrow e$$

We note that this model has been used successfully to capture far more than its simple description implies. In particular, it has been used to model threads, semaphores, concurrent objects, *etc.*

2.2 The Fidge/Mattern Timestamp

The Fidge/Mattern vector timestamp was designed as a fully-distributed algorithm, in which vectors are appended to messages within the computation. In our context these timestamps are computed centrally in the monitoring entity, rather than within the parallel computation. As such, their calculation is somewhat different from that described by Fidge and Mattern. First, each process p is assigned a unique identifier p_i, where $0 < p_i \leq N$ and N is the number of processes in the computation. Each event e is then assigned a vector timestamp $\mathcal{FM}(e)$ of size N as follows. Define $\mathcal{FM}'(f)$ for event f, thus:

$$\mathcal{FM}'(f)[i] = \begin{cases} \mathcal{FM}(f)[i] + 1 & \text{if } i = p_f \\ \mathcal{FM}(f)[i] & \text{otherwise} \end{cases} \quad (1)$$

where $\mathcal{FM}(f)$ is the Fidge/Mattern timestamp of event f and p_f is the process in which event f occurred. Then the Fidge/Mattern timestamp of event e is the element-wise maximum of $\mathcal{FM}'(f)$ of all events f that are immediate predecessors of e:

$$\mathcal{FM}(e) = \max_{f <: e} \left(\mathcal{FM}'(f) \right) \quad (2)$$

where $f <: e$ denotes f is an immediate predecessor of e and $\max$ is an element-wise maximum computation. The precedence test is then:

$$e \rightarrow f \iff \mathcal{FM}(e)[p_e] < \mathcal{FM}(f)[p_e] \quad (3)$$

An example of the Fidge/Mattern timestamp is shown in Figure 2.

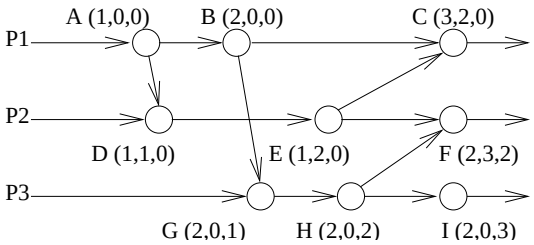

Figure 2. Fidge/Mattern Timestamps

2.3 Self-Organizing Hierarchical Cluster Timestamps

The self-organizing hierarchical cluster timestamp is based on the idea of grouping processes into clusters. Clusters in turn are grouped hierarchically into clusters of clusters, and so on recursively, until one large cluster encompasses the entire computation. It must be clearly emphasized at this point that these clusters are not in any way under the control of the user, nor do they (explicitly) represent any clustering that is a part of the parallel computation. Rather, these clusters are simply a mechanism by which processes are grouped with the intent of creating more efficient vector timestamps.

The efficacy of this timestamp is based on two observations. First, events within a cluster can only be causally dependent on events outside that cluster through receive events from transmissions that occurred outside the cluster. Such events are called "cluster receives." By identifying such cluster-receive events, it is possible to shorten the timestamp of all other events within the cluster to the number of processes in the cluster. Second, in many parallel and distributed computations, most communication of most processes is with a small number of other processes. If the clusters capture this communication locality, then there should be few cluster-receive events, and so the average space-consumption per timestamp will be significantly less than with the Fidge/Mattern timestamp.

Cluster-timestamp creation is the point at which the clustering strategy intersects with the timestamp algorithm. We therefore briefly describe the timestamp-creation algorithm. The algorithm first computes the Fidge/Mattern timestamp for the event. If the event is not a cluster receive, it is assigned a cluster timestamp that is the projection of the Fidge/Mattern timestamp over the processes in the cluster. If the event is a cluster receive, the algorithm acts according to whether the two clusters may be merged or not. This determination of mergeability is a function of the clustering strategy chosen, and is thus the point of intersection of the two algorithms. We discuss it further in Section 3. For events that are non-mergeable cluster receives, the timestamp assigned is the Fidge/Mattern timestamp. A note is made that this is the greatest cluster receive within this process at this point. For an event that is a mergeable cluster receive, the algorithm merges the clusters, thus making the event no longer a cluster receive. The cluster timestamp assigned is the projection of the Fidge/Mattern timestamp over the processes of the newly-merged cluster. The algorithm deletes Fidge/Mattern timestamps that are no longer needed.

The space consumption of this algorithm is $O(N)$ for cluster-receive events, where N is the number of processes in the computation. All other events have a timestamp size of $O(c)$, where c is the number of processes in the cluster (hereafter referred to as the cluster size). Clearly the efficacy of the algorithm is a function of the ability of the clustering strategy to minimize the number of cluster receive events.

2.4 Related Work

Finally, before describing the clustering strategies we have employed, we briefly summarize related approaches to the problem of efficient precedence determination, and in particular, methods for space-reduction for encoding partial orders.

There are various other approaches that have been taken to the problem of reducing the size of vector timestamps. Fowler and Zwaenepoel [4] create direct-dependency vectors. While these vectors can be substantially smaller than Fidge/Mattern timestamps, precedence testing requires a search through the vector space, which is in the worst case linear in the number of messages. Jard and Jourdan [9] generalize the Fowler and Zwaenepoel method, but have the same worst-case time bound. Singhal and Kshemkalyani [18] take the approach of transmitting just the information that has changed in the vector between successive communications. While not directly applicable in our context, it is possible to use a differential technique between events within the partial-order data structure. However, when we evaluated such an approach we were unable to realize more than a factor of three in space saving. Ward [19] has an approach based on Ore timestamps [16], though it is only applicable to low-dimension computations. There is also some amount of work on dynamic transitive closure (*e.g.*, [11]), though the problem being addressed is more general than is our problem, and the results are not as good for our application.

More recently, Garg and Skawratananond developed a technique for timestamping synchronous computations [5]. Their method results in timestamps that have size equal to the vertex cover of the communication graph of the computation. Since the communication graph is rarely known *a priori*, this technique would in practice only be applicable as a static algorithm, and then only for synchronous systems. Finally, their method requires unary events to have timestamps that are twice the size of those for synchronous events, and the timestamp for such unary events cannot be fixed until a subsequent synchronous event occurs in the same process.

3 Clustering Strategies

We now describe the clustering strategies we employed. While the approaches are fairly standard (see Kaufman and Rousseeuw [10] for various clustering methods), there are three significant issues that we focus on. First, we discuss the algorithm choice. Second, we indicate how we measure similarity and dissimilarity between processes. Finally, we specify how we deal with synchronous events.

3.1 Static Clustering

We initially considered and implemented variations on the k-means and k-medoid methods. We found, however, that the problem with the k-means approach was that determining a centroid is not obvious when dealing with communication events between processes. There is no clear definition for an abstract process that represents the centre of a cluster of processes. The k-medoid method was

similarly problematic, in that the clusters created required one process to specifically be considered to be the central process for the cluster. Again, this does not match the reality of parallel computations well. Finally, the results of these approaches were poor, as they select the number of clusters to be created, rather than bounding the size of the desired clusters. The effect was that many processes were grouped within a single cluster, while the remaining clusters were sparse. The result of such a clustering would be that the cluster-timestamps would have little benefit over Fidge/Mattern timestamps, as most events would have a timestamp that was little smaller than the Fidge/Mattern one.

We therefore adopted a hierarchical clustering method. The reader should be careful to distinguish the hierarchical cluster approach used here from the hierarchical cluster-timestamp algorithm. The cluster-timestamp algorithm has a hierarchy of clusters, though in this paper, we are just exploring two levels of clusters. The hierarchical clustering method, as we use it here, produces a single level of clusters. It does so by progressively merging smaller clusters into larger ones. The number of processes that are permitted in any given cluster is limited to a maximum cluster size, for the reason described above.

To describe the precise algorithm we implemented, we first describe our method for determining similarity, and how we deal with synchronous events. Since we are concerned with minimizing the number of cluster-receive events, this is the key element for deciding to merge two clusters. We therefore pairwise compare all current clusters to determine the number of communication occurrences between the clusters. There is a communication occurrence between two clusters if there is a send event in one cluster and its corresponding receive event is in the other cluster. In practice, we examine the receive events in each cluster in the pair, and check to see if the corresponding send event is in the other cluster of the pair. This count represents the number of cluster-receive events that will be required if the cluster pair in question are not merged.

A naive approach at this point would be to simply select that pair of clusters that had the greatest pairwise communication and to merge those two clusters (subject to the constraint of limiting the maximum cluster size). This is probably a poor choice. The problem with this approach is that as clusters increase in size, they are likely to have more communication with other clusters, purely by virtue of their size. Rather than take this approach, we normalize the communication count based on the combined size of the pair of clusters. We then select the cluster pair with the highest normalized communication count for merging (again, subject to the maximum cluster size limit), thus taking a greedy algorithm approach. The reader should note that considering all possible combinations of processes in clusters would be computationally too expensive to perform.

```
 1: do
 2:    CRMax ← 0
 3:    c_1^m ← 0
 4:    c_2^m ← 0
 5:    ∀c_i ∈ clusters
 6:       ∀c_j ≠ c_i ∈ clusters
 7:          if ((|c_i| + |c_j|) > maxCS) continue
 8:          else
 9:             CR_ij ← communication (c_i, c_j)
10:             CR ← CR_ij/(|c_i| + |c_j|)
11:             if (CR > CRMax)
12:                CRMax ← CR
13:                c_1^m ← c_i
14:                c_2^m ← c_j
15:    clusters ← clusters − c_1^m
16:    clusters ← clusters − c_2^m
17:    c_3 ← c_1^m ∪ c_2^m
18:    clusters ← clusters ∪ c_3
19: while (CRMax > 0);
```

Figure 3. Static Clustering Algorithm

Finally, we note that synchronous events present a small additional complication. A synchronous event is effectively both a transmit and a receive. Thus, each synchronous communication is treated as two communication occurrences, rather than a single communication occurrence, since the benefit of merging the two clusters in question would be to remove two cluster-receive events, not one.

The formal algorithm we implemented is as shown in Figure 3. Lines 2–4 set the cluster pair that has maximum similarity to the unselected pair. In particular, line 2 sets the current maximum normalized cluster-receive count to zero. If it remains zero during the pairwise comparison, the algorithm terminates (line 19), as this indicates that there is no pair of clusters that can merge (*i.e.* the resultant cluster does not exceed the maximum cluster size permitted ($maxCS$)) that have a communication occurrence between them.

Lines 5–14 perform the pairwise comparison to select the most similar pair of clusters. Specifically, lines 5 and 6 form the nested loop over all clusters. Line 7 automatically rejects any cluster pair whose merger would exceed the maximum permitted cluster size, $maxCS$. Line 9 determines the number of communication occurrences between the cluster pairs for those cluster pairs whose merger is permitted by the cluster-size constraint. Line 10 normalizes that communication count, while line 11 determines if it exceeds the current best pair. If it does, this pair is set as the current best pair (lines 12–14).

After all pairwise comparisons have been performed, the pair of clusters with the most similarity is known. Lines 15–16 remove that pair from the set of clusters, and add in the

new merged cluster that resulted from merging the selected pair (line 17–18).

The algorithm terminates when there are no two clusters that can merge, because the size restriction prevents it or there is no communication occurrence between two mergeable clusters. We follow this approach because no matter how poor a cluster might seem, if it reduces the number of cluster receives at all, it is better than if it were not formed. In particular, we presume that any implementation of the cluster-timestamp algorithm will use vectors of size equal to the maximum cluster size, since any variation in sizing of the vectors is likely to have a detrimental impact on the performance of the memory-allocation system.

Finally, we note that the algorithm is initialized by placing each process within its own cluster. The algorithm will operate in $O(N^3)$ time, as the outer while loop can iterate at most N times, since each iteration will reduce the number of clusters by one. Since this is a static algorithm, this performance is acceptable. Further, when implemented, we observed that the performance was more than sufficient. The results for this algorithm are presented in Section 4.

3.2 Dynamic Clustering

A static clustering implies the overall timestamp algorithm must be static. Indeed, a simple approach wherein the static clustering algorithm might be used with the hierarchical cluster timestamp would be to perform two passes over the event data. The first pass would cluster the data, and the second pass would timestamp it.

In general such a static timestamp algorithm is not sufficient to the task of parallel-program communication visualization. Rather, what is required is a dynamic algorithm, since the observation tool is expected to operate while the computation is executing. The hierarchical cluster timestamp is dynamic, provided that it is paired with a dynamic clustering strategy. There is currently only one dynamic clustering strategy that has been explored for that timestamp, namely the merge-on-1$^{\text{st}}$-communication approach. There are two key problems with that algorithm. First, while it is capable of producing excellent space reduction over Fidge/Mattern timestamps, it is only capable of sodoing if the maximum cluster size is selected appropriately. The problem with such a selection is that is it not known *a priori* what is an appropriate value for this parameter. It is only after execution of the algorithm that it is known if it was a good choice or not. When a poor value is chosen, the space consumption can approach that of Fidge/Mattern timestamps. As such, the algorithm cannot be reliably used as a dynamic timestamp algorithm.[2] The second problem, which is in effect the cause of the first problem, is that there is no single value, or range of values, of maximum cluster size that is appropriate for all computations.

As a result of these problems we have started to investigate other dynamic clustering strategies. In this paper we investigate the merge-on-N$^{\text{th}}$-communication approach, as it is a fairly natural extension of the merge-on-1$^{\text{st}}$-communication approach. Rather than merge on first communication between clusters, we keep a matrix that identifies the total number of cluster receives that have occurred thus far between the two clusters. This is then normalized by the size of the clusters, as was the case in the static algorithm. The decision to merge is then made when this normalized cluster-receive value passes a threshold. The algorithm thus degenerates to merge-on-1$^{\text{st}}$-communication if that threshold is set to 0. We now present our results for these two algorithms.

4 Experimental Results

We have evaluated our algorithms over more than 50 different parallel and distributed computations covering a variety of different environments, including Java [7], PVM [6] and DCE [3], with up to 300 processes in each computation. The PVM programs tended to be SPMD style parallel computations, and included the programs from the Cowichan benchmark [23]. A number of them exhibited close neighbour communication and scatter-gather patterns. The Java programs were web-like applications, including various web-server executions. The DCE programs were sample business-application code. While space limitations prevent the publication of the code used, it is available for evaluation, together with the complete raw results, on our website at http://www.ccng.uwaterloo.ca/~pasward/-ClusterTimestamp.

For our experiments we compared the space requirements for four algorithms: Fidge/Mattern timestamps, cluster timestamps using merge-on-1$^{\text{st}}$-communication, cluster timestamps using the static clustering algorithm described in Section 3.1, and cluster timestamps using the merge-on-N$^{\text{th}}$-communication algorithm described in Section 3.2. We cannot compare our work with the Garg and Skawratananond algorithm as their method is limited to synchronous computations and none of our computations contain exclusively synchronous communication. The comparison to Fidge/Mattern timestamps is appropriate as existing observation tools use it.

With the exception of the Fidge/Mattern algorithm, all of the algorithms under comparison have a single tunable parameter: the maximum cluster size. We therefore varied this value from 2 to 50 processes and observed the ratio

[2] The algorithm can, however, be used in a static mode. The approach taken is to iterate over the event data multiple times, using different values of maximum cluster size, until one is found that provides a suitable space saving. It should be noted, however, that there is no guarantee that such a cluster-size choice will be found.

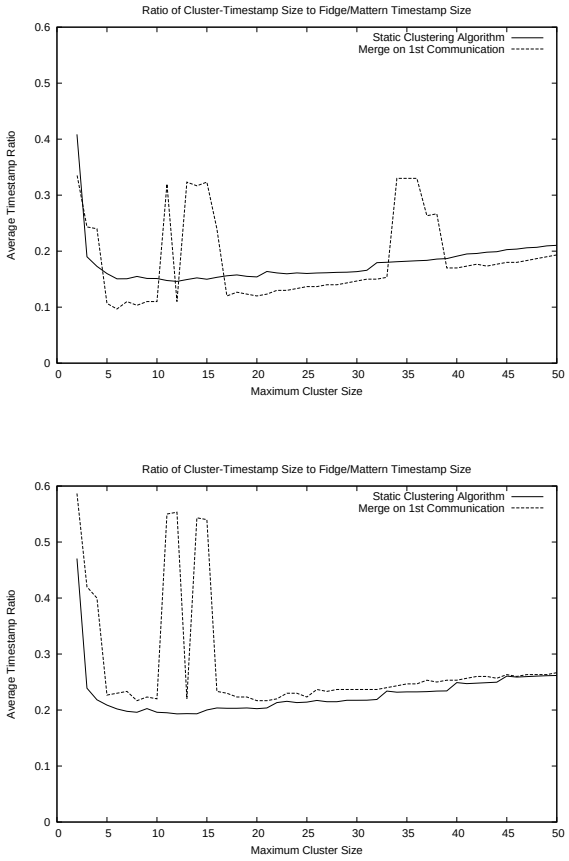

Figure 4. Ratio of Static Cluster to Fidge/Mattern Sizes

of the average cluster-timestamp size to the Fidge/Mattern-timestamp size. We followed the POET and OLT approach of assuming that the observation tool encodes timestamps using a fixed-size vector. By default this is 300. The cluster timestamps were assumed to be encoded using a vector of size equal to the maximum cluster size. These assumptions are consistent with the current behaviour of existing observation tools.

A couple of sample results for the static cluster algorithm for two distinct computations are shown in Figure 4. The figure shows the ratio of the cluster timestamp size to the Fidge/Mattern timestamp size. It illustrates the effect of two clustering algorithms, the static one and the merge-on-1st-communication, used in the initial evaluation of the cluster-timestamp algorithm. The Fidge/Mattern timestamp would have a ratio of 1, and is thus not shown in the figure since it is off the scale. These results are typical of the behaviour we observed for the various computations.

Several comments must be made about these results. First, we note that the static algorithm does not always produce the least space consumption. In particular, as can be seen in the upper figure (which shows the worst case we observed), in some instances the static algorithm was as much as 5% worse than the merge-on-1st-communication approach. We note that this is a small space-cost difference, and not relevant for the purpose of this paper. What is relevant is that these results clearly illustrate our second claim, namely that there exist cluster algorithms that are not significantly sensitive to the maximum cluster size selected. The static clustering algorithm we created produces relatively smooth ratio curves, demonstrating that it does not have the sensitivity to maximum cluster size that merge-on-1st-communication exhibits.

A couple of sample figures do not necessarily show the whole picture. We therefore examined all computations over the three different environments. In doing so we found that if the maximum cluster size is any value between 9 and 17 (inclusive) the resulting timestamp was within 20% of the best timestamp size achieved for all but one computation. This clearly demonstrates our second claim over all of our extant event data. We must emphasize at this point that such a range of maximum cluster sizes that produce an acceptable space reduction simply does not exist for either the merge-on-1st-communication strategy or for fixed contiguous clusters. Further, we must reiterate that this maximum-cluster-size parameter must, in general, be selected prior to the timestamping, and thus this lack of such an ideal in the merge-on-1st-communication case renders that technique of limited value.

To confirm our first claim, for all computations studied, we computed the range of maximum cluster sizes for which the timestamp size was within 20% of the best timestamp size achieved. The result of this was that we determined that for all computations a cluster size of 13 or 14 resulted in a timestamp size that was within 20% of the best achievable. By way of comparison, in Ward's analysis [20] of merge-on-1st-communication it was observed that there was no single maximum cluster size that was suitable for all computations. Indeed, for all but a couple of cases, less than 80% of the computations were within 20% of the best for any given maximum cluster size.

The results of our static clustering algorithm must be emphasized and reiterated: this work has shown that there exists a maximum cluster size that will result in cluster timestamps whose size is within 20% of the best size. Further, there is a significant range over which the vast majority of computations exhibit very good timestamp size reduction as compared with the Fidge/Mattern timestamp. Ward's original work on this timestamp algorithm did not demonstrate that such an ideal maximum cluster size existed, and in fact pointed strongly to the possibility that such an ideal did not exist. These results for the static clustering algorithm clearly demonstrate that this ideal does in fact exist, and

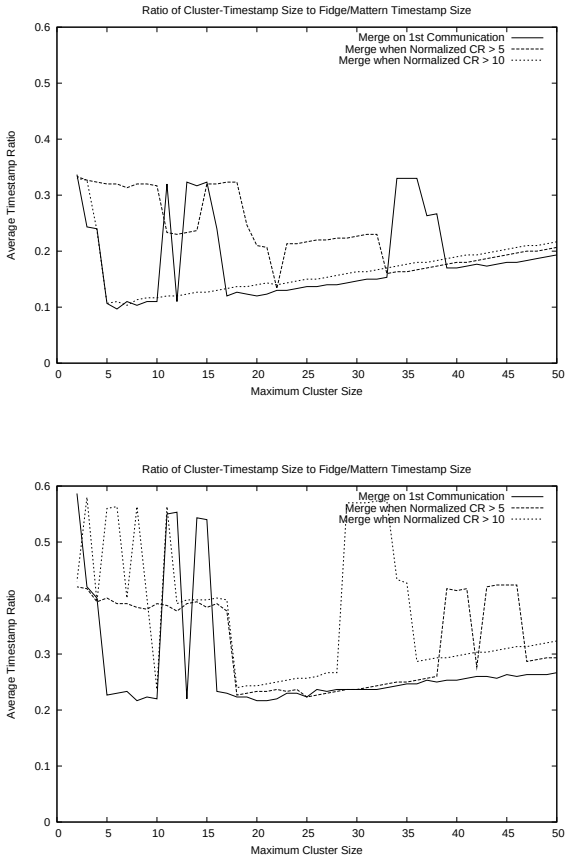

Figure 5. Ratio of Static Cluster to Fidge/Mattern Sizes

that it is simply a question of developing a suitable dynamic clustering strategy so as to capture that ideal.

With this in mind, we now turn our attention to the dynamic clustering algorithm. In this case we expected the results to be a gradual smoothing of the ratio curve, from that of merge-on-1st-communication. In addition, we expected the overall curve to rise, as the number of events that needed full Fidge/Mattern timestamps would increase because cluster merging was being deferred.

Sample results using the dynamic clustering algorithm are presented in Figure 5. The sample computations represented in this figure are the same as those computations shown in Figure 4. Several observations can be made about these results. First, we note that the upper graph indicates that this dynamic clustering method offers a lot of promise. In particular, we note that as the normalized cluster-receive threshold increased, the result was indeed the flatter curve that we had hoped for. More surprisingly, however, is the fact that the curve is not substantially higher than the merge-on-1st-communication curve at its best. This appears to be because the computation has a very large number of events, but the number of cluster-receive events becomes fixed once the maximum cluster size reaches 22. This may be because the algorithm ceases to cluster processes, as the threshold is too large.

The second point of note is that the algorithm does not always smooth the curve as much as might be desired, and sometimes when it does so, it smooths it at a significantly higher timestamp size. Thus, the lower figure clearly exhibits the problem that, while smoothed at the CR > 5 point, it is smoothed at the 40% mark, not the 20% mark. Further, as the merging criteria was raised, the curve became less predictable. It is clear, therefore, that more work is required to achieve a suitable dynamic clustering algorithm.

As with the static algorithm, sample results do not present the full picture. We therefore looked at the raw results to determine over what range of maximum cluster size the computations exhibited a timestamp space reduction that was within 20% of the best achieved. We did this for a normalized cluster-receive threshold of 10, since that appeared to be the most promising based on visual inspection of the graphs. We were not able to find such a range that covered all computations for this dynamic clustering algorithm. However, we did find that when the maximum cluster size permitted was between 22 and 24 processes (inclusive), all but two computations had a timestamp size that was within 20% of the best size. The two that exceeded 20% of their best size over that range still had an average timestamp size that was less than one-third of their Fidge/Mattern timestamp size.

The full raw-result information for our experiments is available on our web site at `http://www.shoshin.uwaterloo.ca/~pasward/ClusterTimestamp`.

5 Conclusions and Future Work

In this paper we have evaluated clustering algorithms for self-organizing hierarchical cluster timestamps. We have seen that a simple static clustering algorithm demonstrates that the cluster-timestamp technique can work well, and is not inherently sensitive to cluster-size selection. Further, we have shown that it is possible to create clustering strategies that work well with a large range of computations, rather than having parameters that are specific to each individual computation.

Third, we have evaluated a simple extension to the dynamic merge-on-1st-communication clustering strategy. We call this extension "merge-on-Nth-communication" though this is a normalized count. This approach allows for the selection of a maximum cluster size that produced very good results for all but two computations, and satisfactory results for those two cases where it was not ideal.

We are therefore investigating two lines of development.

First, we are exploring alternate dynamic clustering strategies. Second, we are developing two variants of the cluster timestamp. The first variant will collect a significant number of events before performing a static clustering and subsequent timestamp operation. Such an approach will require a mechanism for precedence determination for those events that have yet to receive a cluster timestamp. The second variant we are examining is one in which processes will be permitted to migrate between clusters in the event that it is apparent that the clustering initially selected is a poor one. We expect to report on these methods in the near future.

Acknowledgments

The authors would like to thank IBM for on-going support of this work, and the various reviewers for their comments, which have helped to improve this paper.

References

[1] G. Eisenhauer and K. Schwan. An object-based infrastructure for program monitoring and steering. In *Proceedings 2nd SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT'98)*, pages 10–20, 1998.

[2] C. Fidge. Logical time in distributed computing systems. *IEEE Computer*, 24(8):28–33, 1991.

[3] O. S. Foundation. *Introduction to OSF/DCE*. Prentice-Hall, Englewood Cliffs, New Jersey, 1993.

[4] J. Fowler and W. Zwaenepoel. Causal distributed breakpoints. In *Proceedings of the 10th IEEE International Conference on Distributed Computing Systems*, pages 134–141. IEEE Computer Society Press, 1990.

[5] V. K. Garg and C. Skawratananond. Timestamping messages in synchronous computations. In *Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems*, pages 552–559. IEEE Computer Society Press, 2002.

[6] A. Geist, A. Begulin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, Massachusetts, 1994.

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996. Available at http://java.sun.com/docs/books/jls/.

[8] IBM Corporation. IBM distributed debugger for workstations. Online documentation available at: `http://-www-4.ibm.com/software/webservers/-appserv/doc/v35/ae/infocenter/olt/-index.html`.

[9] C. Jard and G.-V. Jourdan. Dependency tracking and filtering in distributed computations. Technical Report 851, IRISA, Campus de Beaulieu – 35042 Rennes Cedex – France, August 1994.

[10] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data : An Introduction to Cluster Analysis*. John Wiley and Sons, 1990.

[11] V. King and G. Sagert. A fully dynamic algorithm for maintaining the transitive closure. In *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing*, pages 492–498. ACM, 1999.

[12] D. Kranzlmüller. *Event Graph Analysis for Debugging Massively Parallel Programs*. PhD thesis, GUP Linz, Linz, Austria, 2000.

[13] D. Kranzlmüller, S. Grabner, R. Schall, and J. Volkert. ATEMPT — A Tool for Event ManiPulaTion. Technical report, Institute for Computer Science, Johannes Kepler University Linz, May 1995.

[14] T. Kunz, J. P. Black, D. J. Taylor, and T. Basten. POET: Target-system independent visualisations of complex distributed-application executions. *The Computer Journal*, 40(8):499–512, 1997.

[15] L. Lamport. Time, clocks and the ordering of events in distributed systems. *Communications of the ACM*, 21(7):558–565, 1978.

[16] O. Ore. *Theory of Graphs*, volume 38. Amer. Math. Soc. Colloq. Publ., Providence, R.I., 1962.

[17] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distributed Computing*, 7(3):149–174, 1994.

[18] M. Singhal and A. Kshemkalyani. An efficient implementation of vector clocks. *Information Processing Letters*, 43:47–52, August 1992.

[19] P. A. Ward. A framework algorithm for dynamic, centralized dimension-bounded timestamps. In *Proceedings of the 2000 CAS Conference*, November 2000.

[20] P. A. Ward. *A Scalable Partial-Order Data Structure for Distributed-System Observation*. PhD thesis, University of Waterloo, Waterloo, Ontario, 2002.

[21] P. A. Ward and D. J. Taylor. A hierarchical cluster algorithm for dynamic, centralized timestamps. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems*, pages 585–593. IEEE Computer Society Press, April 2001.

[22] P. A. Ward and D. J. Taylor. Self-organizing hierarchical cluster timestamps. In R. Sakellariou, J. Keane, J. Gurd, and L. Freeman, editors, *EuroPar'01 Parallel Processing*, volume LNCS 2150 of *Lecture Notes in Computer Science*, pages 46–56. Springer-Verlag, August 2001.

[23] G. V. Wilson and R. B. Irvin. Assessing and comparing the usability of parallel programming systems. Technical report, University of Toronto, 1995.

[24] Y. M. Yong. Replay and distributed breakpoints in an OSF DCE environment. Master's thesis, University of Waterloo, Waterloo, Ontario, 1995.