Comparison of Message Aggregation Strategies for Parallel Simulations on a High Performance Cluster

C. D. Pham
RESAM, Université Lyon 1, Claude Bernard
Bât ISTIL, 43 bd du 11 novembre 1918
69622 Villeurbanne Cedex - France
email:cpham@lhpca.univ-lyon1.fr

Abstract

Parallel simulations of fine grain applications usually generate a large amount of messages. The overhead for sending these messages over an interconnection network can dramatically limit the speedup of a parallel simulation. In this case, message aggregation techniques can increase the granularity of the application and reduce the communication overhead. This paper compares sender-initiated and receiver-initiated message aggregation strategies and addresses their performance when applied to parallel simulations of high-speed communication network models on a high performance cluster. Receiver-initiated strategies were initially motivated by the cyclic behavior of the parallel simulation kernel and by the capabilities of the test-bed environment to handle large messages up to 2000 bytes with small overheads.

1 Introduction

Parallel simulations of communication network models usually require a large amount of events to be simulated as these events typically represent packets in the real system. High-speed network models emphasize this property as the number of messages increases dramatically. It is therefore very desirable to increase the granularity of the application and to minimize the overhead for communication among the processors, especially on a cluster of workstations. These optimizations can be performed at the lowest level in the communication software [4] or in the application at a higher level. This paper focuses on application-level optimizations and more precisely on message aggregation that groups several small messages for the same destination into a single larger one [5, 3].

Message aggregation is based on the observation that it is less costly to send an M byte message than to send n times an m byte message with $n * m = M$. For fine-grain applications message aggregation can become very efficient thus motivating experiments in this area. For instance, dynamic message aggregation policies have been used with a Time Warp based simulator in [3]. Results show that application-level message aggregation can reduce the simulation time by around 30% in the best case on an Ethernet-based cluster. Investigating the use of high performance clusters in [8] (3 Pentium Pro interconnected by Fast Ethernet and Myrinet), the authors reported that message aggregation presents disappointing results for Myrinet (with MPI-BIP) and encouraging results for Fast Ethernet. However, it has been shown that message aggregation can become efficient on a high performance cluster (Myrinet-based) providing that the behavior of the lowest level of communication is taken very carefully into account [6].

In [6] the constraint put on the aggregate size was to remain below the 156 bytes threshold set by the communication layer which moves from a small to a large message sending strategy. Recently, this limit has been increased to 2000 bytes opening new opportunities for large aggregate size. Therefore it became possible to imagine a scheme where the sender always aggregates as much as possible: this was the motivation for receiver-initiated strategies, as opposed to the commonly used sender-initiated strategy [3, 6, 8], where the sender would keep aggregating until the receiver explicitly asks for new messages. This paper presents a receiver-initiated message aggregation strategy and compares its performance with the sender-initiated strategy previously described in [6]. These mechanisms are used in the CSAM framework for simulating large communication network models on high performance clusters. The paper is organized as follows: Section 2 briefly presents the parallel simulation application and the parallel simulator's kernel. Section 3 presents the message aggregation strategies. Section 4 presents the test-bed environment and the experimental results. Conclusions are given in Section 5.

2 The parallel simulation application

The application consists of 78 ATM switches (Asynchronous Transfer Mode technology) and traffic sources. The system includes a routing algorithm with dynamic routing functions that provide load-balancing routing features. Simulating high-speed network models at the cell-level requires a large number of small messages to be exchanged between processors (as opposed to more traditional high-performance computing applications where the computation part is larger). It is not unusual to send several millions of messages per minute of wall clock time! Depending on the number of processors used, these events must be sent over the interconnection network with high communication overheads.

The parallel simulator is written in C++ and follows the conservative rules as initially defined in [2]. A conservative method was chosen because the application exhibits good lookahead with the link propagation delay. The life cycle of the simulated objects in the parallel simulator basically consists in processing internal messages and generating messages. Internal messages are taken from the local Future Event List (FEL) and processed in time stamped order. Generated messages that have to go off the processor are sent to the destination using the communication system, otherwise they are directly inserted as new internal events. When an LP has no more eligible internal messages to process, i.e. no more messages in the local event list or it has reached the upper bound of the time window (*safe time*), it would send out a null-message with timestamp *safe time+lookahead*.

3 Message aggregation

Usually the latency for sending a message can be split in a constant part and a size-dependent part. Since the constant part is relatively high, sending several small messages is much more costly than sending a single larger one.

In most fine grained applications the polling for new external events is done periodically and not after every event processing. Our conservative kernel computes a safe upper bound timestamp for event processing and only polls for remote messages at the end of an event processing cycle, thus motivating message aggregation techniques. In the following sub-sections both sender-initiated and receiver-initiated aggregation strategies will be described. For the receiver-initiated strategy, several ways to request an aggregate will also be presented.

3.1 Sender-initiated message aggregation

Sender-initiated message aggregation is the most commonly used version of message aggregation where the sender decides to aggregate messages and to send them in one piece when necessary. The decision to send an aggregate may be based on a timer, a maximum aggregate size, or on the presence of special messages (null-messages for instance).

The sender-initiated strategy described in [6] performs message aggregation with a fixed aggregate size. Let S_{max} be the maximum size allowed for an aggregate: messages from the application are stored in temporary buffers, one per destination processor. When a new message needs to be sent, the final aggregate size S is computed and checked against S_{max}. If $S > S_{max}$ then the current aggregate is sent and the new message initializes a new aggregate. If $S < S_{max}$ then the new message is added to the current aggregate. Then, if $S > S_{max} - min\_event\_size$ the aggregate containing the new message is sent immediately to avoid useless waiting as there will be no more space left for any additional message. Events of size equal to min_event_size can be either a null-message or a model routing control message.

Rules for sending aggregates are broken when the conservative kernel sends null-messages: these special messages immediately trigger the sending of an aggregate, whatever the aggregate size is in order to propagate the simulation time as fast as possible. Note that since null-messages are sent at the end of every event processing cycle, an aggregate is guaranteed to be sent when the LP has no more events to process. This strategy will be noted AGGRxxx where xxx represents the maximum size allowed for an aggregate (in bytes).

3.2 Receiver-initiated message aggregation

Receiver-initiated message aggregation means that the receiver explicitly requests that aggregates are sent to him. This strategy has many nice features because the sender can safely aggregate as much as it can. Also, since LPs process their events in cycle, there is no point to receive aggregates too early because they will not be retrieved from the communication buffer until the end of a cycle. Therefore, it may be interesting to simply let the consumer control the aggregation process. There are several solutions to implement a receiver-initiated mechanism. The implemented solution is to define 2 message tags: DATA and AGGRCTRL tags. Everything that is not an aggregate request message is tagged with DATA.

At the sender side, in order to react very quickly to an aggregate request, an LP periodically polls for AGGRCTRL messages. This task is performed by the simulator's kernel before processing an event. The aggregate request message carries the identity of the sender so that the LP knows which buffer it has to send (there are multiple aggregate buffers, one per destination). The sending of the aggregate is done immediately after the reception of the aggregate request message. Note that the rule the aggregation of a null-message still makes the aggregate to be immediately sent. In a receiver-initiated scheme, it is possible to keep aggregating after a null-message but provision to avoid deadlock must be implemented (because every LP will keep aggregating and null-messages will remain trapped).

At the receiver side, an LP sends an aggregate request either at the end of a processing cycle or at some time during a processing cycle.

3.2.1 Requesting aggregates at the end of each cycle

An aggregate request is sent at the end of a processing cycle when no more eligible messages are available and no new aggregates have arrived in the communication buffers (remember that the sender can still send aggregates because of null-messages). Then the LP would wait in a blocking receive statement for new aggregates to arrive. Aggregate request messages are sent to all possible source of messages.

Since the receiver has to wait, there is obviously a non productive period of time. However, this behavior allows the sender to aggregate messages as much as it can before sending them in one piece. As aggregating more messages reduces the send operation overhead at the sender side (with regard to the latency), this may justify the extra waiting time at the receiver side. This strategy will be noted R_AGGRxxx.

3.2.2 Requesting aggregates in advance

The previous scheme waits until a cycle is completed before requesting new aggregates. It may be interesting to request aggregates in advance in order to get them ready at the end of a cycle. This is the purpose of the optimization described in this section: an LP sends an aggregate request message not at the end of a cycle but when it detects that there are only B safe events left for processing. We will note this strategy RA_AGGRxxx.

The difficulty is to know how many events are still eligible for processing in order to know when only B of them remain. Since all events are stored in the FEL, it is not trivial to know how many events are below the *safe time* without searching through the event list at the beginning of each processing cycle. The objective is to do it with the less possible overhead. We implemented an algorithm based on a fixed size circular lookup table where each entry records the number of events at a given integer timestamp. It is therefore possible to approximate the number of safe events that remain to be processed in order to send an aggregate request message when only B events are left for processing. The choice of B will be discussed later on in the result section.

4 Experimental result

All simulations of the network model have been performed on a cluster consisting of 4 nodes interconnected by a Myrinet switch [1]. Each node is an Intel Pentium Pro 200 MHz with 64MBytes of RAM and a 440FX chip-sets. The myrinet board contains a LANai 4.1 and 256Kbytes of memory. MPI on top of BIP, an enhanced low-level API for Myrinet, is used for the communication layer. The test-bed is provided by the LHPC, a cooperation between ENS-Lyon and Matra Système Information. The network model is simulated for 200000 time slots that represent 0.124s of the real system. The sequential simulation time is 141s and the unaggregated parallel simulation time is 224s, meaning that without aggregation, the parallel simulation performs very poorly.

4.1 Sender initiated, AGGRxxx

Figures 1, 2, 3 and 4 show the AGGR2000 aggregate size distribution for each of the 4 processors. Figure 5 shows the resulting data gathered from all the processors.

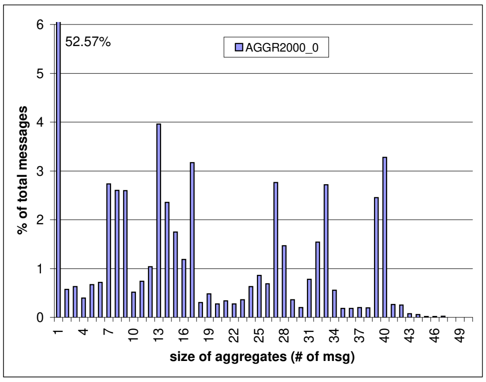

Figure 1. AGGR2000_0

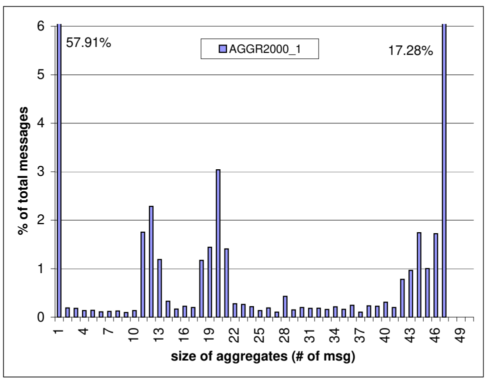

Figure 2. AGGR2000_1

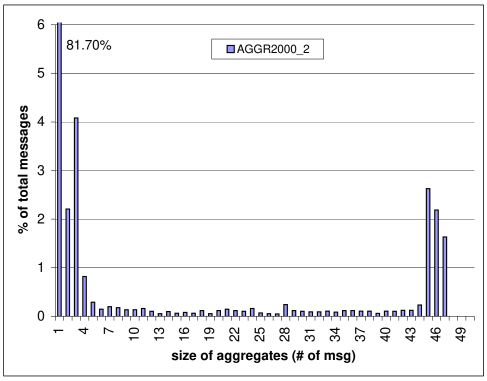

Figure 3. AGGR2000_2

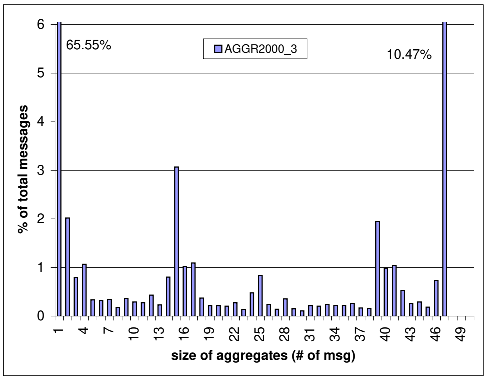

Figure 4. AGGR2000_3

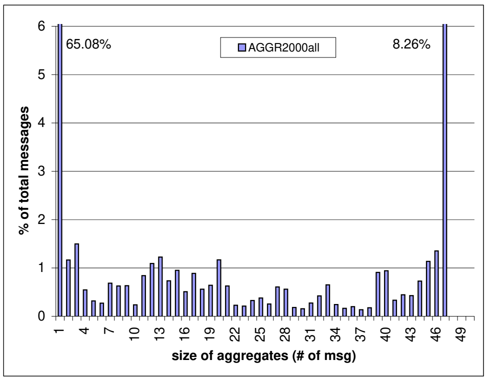

Figure 5. AGGR2000_all

The x-axis represents the aggregate size in number of messages and the y-axis shows the percentage of total messages for a given aggregate size. For readability purposes, items which are above the y-axis limit are shown truncated and only the value is shown on the graph.

The distribution is quite inhomogeneous from one processor to an other. The graph for processor 0 is shaped quite differently from the graph of the other processors. Trying to find a correlation between the number of null messages sent and the shape of the graph, we noticed that processor 0 sent a much smaller number of null-messages than the others. The relation between null-messages and the size of an aggregate is the following: if there is a large number of null-messages then it is more likely that aggregates will be small because null messages break aggregates. Nevertheless, the common points in the graphs are: (i) the large proportion of null-messages sent alone (1-message aggregate) and, (ii) there is always a large proportion of aggregates at the 2 end-points of the curves. In-between, it is unpredictable and the pattern depends on the application. The fact that there is a peak for large aggregate sizes shows that the length of the processing cycles changes considerably during the simulation and can be fairly large. Note that the more messages are aggregated at the sender side, the longer a cycle will be at the receiver side.

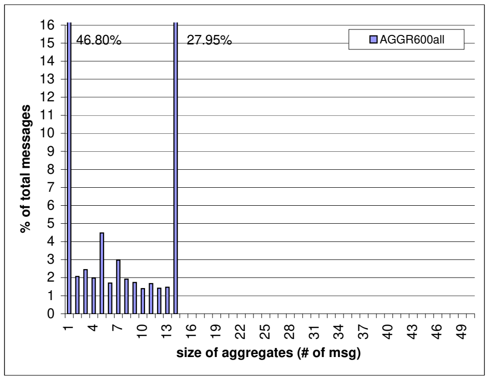

Figure 6. AGGR600all

Figure 6 shows the AGGR600 aggregate size distribution gathered from all the processors. The observations done for AGGR2000 are still valid for this case but transposed to a much smaller range of aggregate sizes. In general, the number of physical messages sent is smaller when the aggregate limit is high. Now, regarding the proportion of null-messages sent alone (1-message aggregates), which is much lower in AGGR600 than in AGGR2000 (figures 6 and 5), the reason is because there are much more physical messages sent in AGGR600 than in AGGR2000 (although the number of null-messages sent in AGGR600 is larger than in AGGR2000 as can be expected).

Figure 7 shows the maximum size (in messages) an aggregate can reach. Almost all cases reach at least once their maximum size (the theoretical size is 4, 9, 14, 19, 29, 36 and 48 respectively with a regular message size of 42 bytes and a null-message size of 24 bytes).

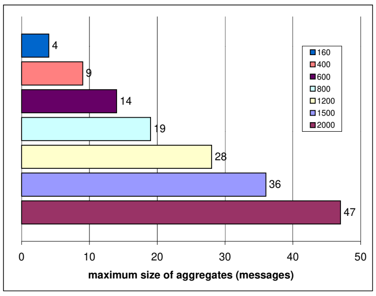

Figure 7. Maximum size of aggregates (messages), AGGRxxx

4.2 Receiver-initiated, R_AGGRxxx

Figures 8 and 9 show data gathered from all processors for R_AGGR2000 and R_AGGR600 respectively.

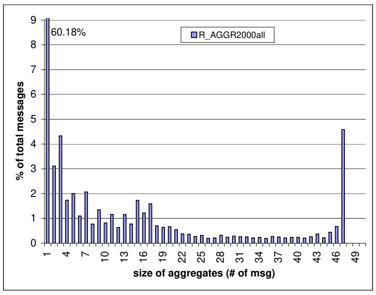

Figure 8. R_AGGR2000

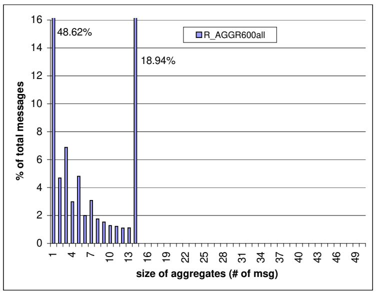

Figure 9. R_AGGR600

In the receiver-initiated version the number of large aggregates is much smaller than in the sender-initiated case. The distribution of aggregate size is denser for aggregate sizes between 1 and 16 messages (comparison between figure 5 and figure 8). Also, the percentage of aggregate reaching the maximum size is smaller (4.65% versus 8.26% for 2000-byte aggregates and 18.94% versus 27.95% for 600-byte aggregates). Regarding the maximum size (in number of messages) an aggregate can reach, the receiver-initiated strategy shows almost identical results compared to the sender-initiated case except that we have 35 and 44 for R_AGGR1600 and R_AGGR2000 respectively instead of 36 and 47 for AGGR1600 and AGGR2000.

Regarding the number of aggregate request messages sent, this traffic represents in all cases about 25% of the total number of physical remote messages. This overhead is not to be neglected, especially when the number of aggregates sent by mean of an aggregate request message is found to be only 2% for R_AGGR600 and 3.6% for R_AGGR2000.

4.3 Receiver-initiated, advance request RA_AGGRxxx

Figures 10 and 11 show the aggregate size distribution for RA_AGGR2000 and RA_AGGR600 (processor 0). The B threshold defined as the number of messages left for processing in the FEL below which an aggregate request is sent is set empirically to 7.

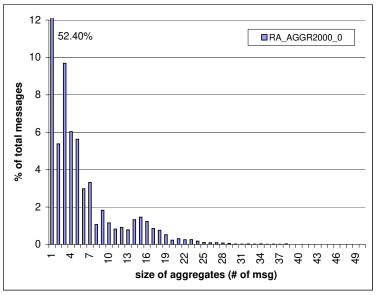

Figure 10. RA_AGGR2000

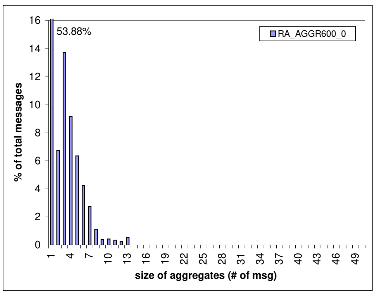

Figure 11. RA_AGGR600

When using advance request, if the maximum size of aggregates is set too small (i.e. 160 or 400 bytes) then the number of events per cycle at the receiver side will be small (less messages per physical message). In this case, advance request not very efficient because the receiver will request aggregates too early. Tests performed with $B = 5$ and $B = 10$ show respectively smaller and larger aggregates.

Setting B higher is not interesting because aggregates have not enough time to be built up. It is quite difficult to find an optimal value for B.

Figure 12 shows the maximum size (in messages) an aggregate can reach with $B = 7$. Advance request makes smaller aggregates when compared to R_AGGRxxx as they are broken earlier by the sender.

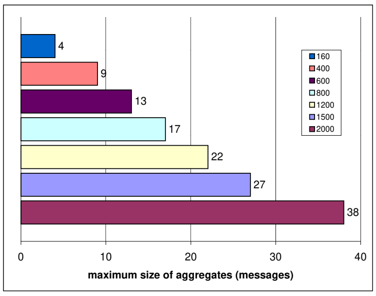

Figure 12. Maximum size of aggregates (messages), RA_AGGRxxx

4.4 Comparison of the aggregate holding time

Figures 13 and 14 show the mean aggregate holding time at the sender side for various aggregate limits (in bytes) for the AGGRxxx and the R_AGGRxxx case respectively. The x-axis indicates in the form of a xy pair which processors are involved (xy means communication between LP x and LP y). As the load between the processors is not well-balanced, the aggregate holding time is not homogeneous from one processor to an other. However, this has little impact on the interpretation of the result. For the sender-initiated case (AGGRxxx), as the aggregate limit increases, we can observe that the mean holding time increases as well, as can be expected. When the aggregate limit is high, e.g. 2000 bytes, the aggregate holding time can be interpreted as the processing cycle duration since aggregates are mainly broken by a null-message sent at the end of a cycle. For an aggregate limit of 2000 bytes, the aggregate holding time increases significantly. If we compare the data from figures 1 to 4 with those in figures 13 and 14, we can see that the processors that show a large aggregate holding time are those that present a high percentage of large aggregates (namely processor 1 and 3).

When comparing figures 13 and 14, we can see that the shape of the 2000-byte curve is quite similar for both the sender-initiated and the receiver-initiated cases. For such a large aggregate limit both strategies have comparable behavior (this will also be shown later on in figures 16 and 17). The additional holding time at the sender side with the receiver-initiated strategy comes from the overhead for

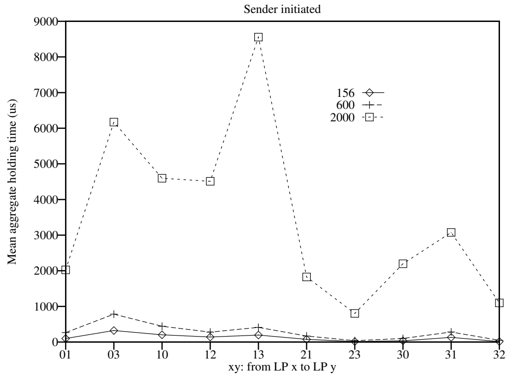

Figure 13. Aggr. holding time, AGGRxxx.

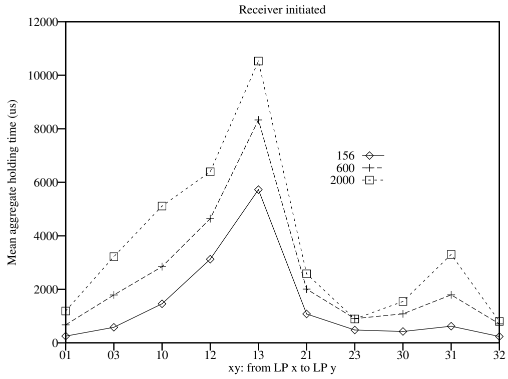

Figure 14. Aggr. holding time, R_AGGRxxx.

polling aggregate request messages. Note that since a receiver is also a sender (and vice versa), the polling overhead affects both sides increasing the processing cycle duration. For smaller aggregate limits (156 and 600 bytes on the curves), R_AGGRxxx shows much larger aggregate holding time, indicating that the sender aggregates longer which is the expected behavior. The RA_AGGRxxx shows very similar results to R_AGGRxxx.

4.5 Impact on the execution time

Figure 15 shows the simulation time (wall clock time) obtained with the various strategies. At this point, we can see that the sender-initiated strategy (AGGRxxx) always performs better and does not need large aggregate limits to be efficient. Note that using larger aggregates does not decrease the performances of the simulator. R_AGGRxxx and RA_AGGRxxx are quite disappointing for small aggregate limits. For large aggregate limits, receiver-initiated strategies perform a little better but still cannot beat the sender-initiated strategy because of the additional overhead introduced by aggregate control request messages.

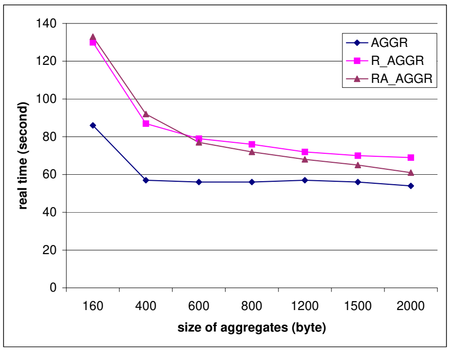

Figure 15. Simulation time for the various strategies.

4.6 Comparison of the receiver idle time

Figures 16 and 17 show the mean receiver idle time (time during which the receiver waits for new messages to arrive) before a new processing cycle for various aggregate limits (in bytes) for AGGRxxx and R_AGGRxxx respectively. For comparison, the mean receiver idle time for the none aggregated version is provided. The x-axis indicates the processor for which the mean idle time is measured.

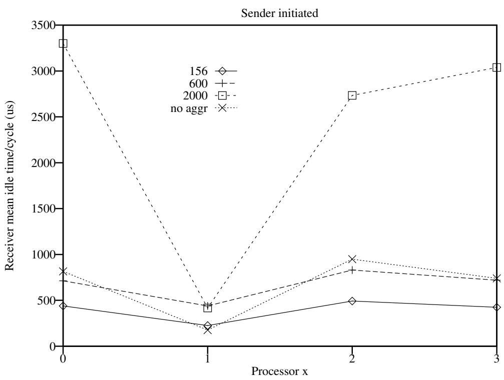

Figure 16. Mean idle time, AGGRxxx

For AGGR156 and AGGR600, all processors except processor 1 show a smaller mean idle time when compared to the unaggregated version. This result is particularly interesting as it shows that aggregation can efficiently take advantage of the cyclic behavior (the receiver only polls new messages at the end of a cycle) of the simulation kernel by increasing the granularity of the application.

For R_AGGRxxx, the mean receiver idle time is larger than that of both the AGGRxxx and the unaggregated version. Since receiver-initiated strategies need an explicit aggregate request, the waiting time is therefore longer compared to the sender-initiated version. Also, we can see that

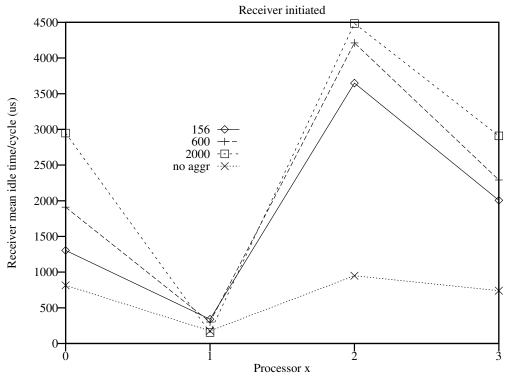

Figure 17. Mean idle time, R_AGGRxxx

the mean receiver idle time is larger for the processors that present a small aggregate holding time. The reason is that a small aggregate holding time usually means a short processing cycle, and therefore a longer waiting time.

Figures 18 and 19 show the percentage of total idle time for a given processor for AGGRxxx and R_AGGRxxx respectively. For comparison, the percentage of total idle time for the none aggregated version is provided.

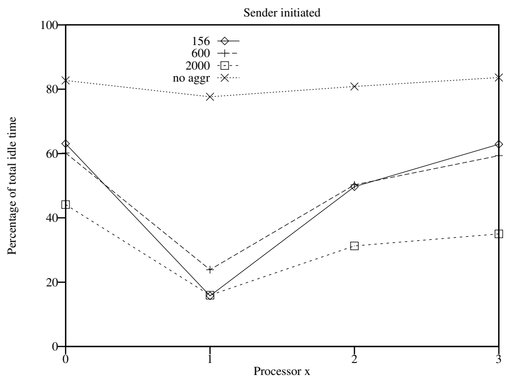

Figure 18. % of total idle time, AGGRxxx.

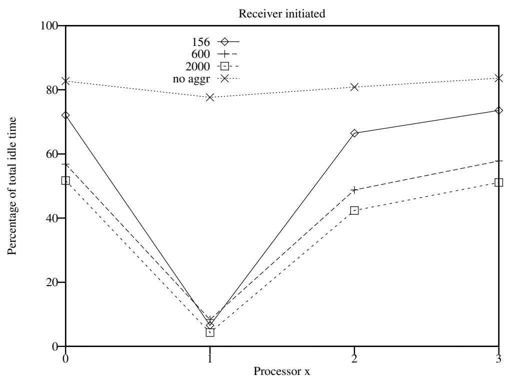

Figure 19. % of total idle time, R_AGGRxxx.

These graphs show the overall impact of message aggregation: aggregation clearly reduces the total idle time of processors by increasing the duration of the processing cycle. The receiver-initiated strategy presents a larger total idle time (except for processor 1) when compared to the sender-initiated strategy that indicates more waiting. If we compare figures 16 and 17 to figures 18 and 19, we can see the benefit of large aggregate limits on the total idle time even though the mean idle time between 2 cycles is larger.

Actually, processor 1 shows the optimal behavior that we would like to generalize to all the processor: large aggregate holding time and small receiver idle time. The reason for this particular result for processor 1 is due to the load imbalance. This processor has luckily a lot of messages to send to all the other processors and also receive a lot of messages from them.

5 Conclusions

Several message aggregation techniques have been presented and applied to parallel simulations of a fine grain communication network model. When comparing sender-initiated and receiver-initiated message aggregation strategies, the former solution always performed better. Receiver-initiated (R_AGGRxxx) succeeded to smooth the aggregate size distribution but failed to provide better performances. The problem typically resides in the number of aggregate request messages that represents 25% of the number of physical messages sent over the network! The advance request strategy (RA_AGGRxxx) performs a bit better for large aggregate limits but still remains below the sender-initiated scheme. For receiver-initiated strategies in general, it seems difficult to reduce the number of aggregate request messages. Using piggy-backing may be possible but this solution arouse many problems for buffer management (priority of piggy-backed messages). This study also reveals the impact of the load imbalance on the processor idle time when message aggregation is used. It would be interesting to add load-balancing features based on the receiver idle time.

References

[1] Boden, N. J., Cohen, D., Feldermann, R. E., Kulwik, A. E., Seitz, C. L., Seizovic, J. N., and Su, W. K. MYRINET: A Gigabit per second Local Area Network. *IEEE-Micro*, 15:29–36, February 1995.

[2] Chandy, K., Misra, J.: Distribution Simulation: A Case Study in Design and Verification of Distributed Programs. *Trans. on Soft. Eng.*, **5**(5) 440–452.

[3] Chetlur, M., Abu-ghazaleth, N., Radhakrishnan, R., Wilsey, P. A.: Optimizing Communication in Time Warp Simulators. *In Proceedings of PADS'98*, Banff, Canada, pp64-71.

[4] Chetlur, M., Sharma, G. D., Abu-Ghazaleh, N., Rajasekaran, U., Wilsey, P. A.: An Active Layer Extension to MPI. *In proceedings of the EuroPVM/MPI '98, (LNCS 1497)*, September 1998, pp97-104.

[5] Gropp, W., Lusk, E.: Tuning MPI Programs for Peak performance. `http://www.mcs.anl.gov/mpi`.

[6] Pham, C. D., Albrecht, C,.: Optimizing Message Aggregation for Parallel Simulation on High Performance Clusters. *In Proceedings of MASCOTS'99*.

[7] Prylli, L., Tourancheau, B.: BIP: A New Protocol Designed for High Performance Networking on Myrinet. *In Parallel and Distributed Processing, IPPS/SPDP'98*. **1388** of *LNCS*, pages 472–485. Springer-Verlag.

[8] Rajasekaran, U. K. V. et al.: Addressing Communication Latency Issues on Clusters for Fine Grained Asynchronous Applications - A Case Study. *In Proc. of the Intl. Workshop on Personal Computer based Network of Workstations (PCNOW'99)*.