Methodology for Adaptive Active Message Coalescing in Task Based Runtime Systems

Bibek Wagle*‡§¶, Samuel Kellar†¶, Adrian Serio‡§‖, Hartmut Kaiser‡*§‖
*School of Electrical Engineering and Computer Science, Louisiana State University, Baton Rouge, USA
†Department of Physics and Astronomy, Louisiana State University, Baton Rouge, USA
‡Center for Computation and Technology, Louisiana State University, Baton Rouge, USA
§The STE||AR Group
¶{bwagle3, skella1}@lsu.edu
‖{aserio, hkaiser}@cct.lsu.edu

***Abstract*—Overheads associated with fine grained communication in task based runtime systems are one of the major bottlenecks that limit the performance of distributed applications. In this research, we provide methodology and metrics for analyzing network overheads using the introspection capabilities of HPX, a task based runtime system. We demonstrate that our metrics show a strong correlation with the overall runtime of our test applications. Our aim is to eventually use these metrics to tune, at runtime, parameters relating to active message coalescing. This method improves on the postmortem analysis techniques that are currently employed to tune network settings in distributed applications.**

I. Introduction

As we move towards exascale computing where tens of thousands of nodes will work together in solving complex scientific problems, task based runtime systems become a viable alternative to the de facto standard of High Performance computing, MPI [1]. The success of task based runtime systems is based on the fact that most algorithms can be decomposed into fine grained units of work that can be executed by the runtime system with very little overhead. A side effect of creating fine grained units of work (tasks) is in fine grained communication patterns when dealing with a large scale distributed application.

Efficient communication across nodes in a cluster is largely dependent on latency and the bandwidth of the network as well as the overheads associated with the creating and sending of messages [2]. If we are sending a large number of messages in quick succession, these overheads rapidly aggregate. In the context of a task based runtime system, where fine grained communication is ubiquitous, efficient use of network bandwidth and reduction of overheads introduced by the transmission of information is vital. Any improvements that can be made in this context have the potential to improve the overall execution time of the distributed application.

Combining many small messages and sending them as a larger message, (see Figure 1) is an optimization technique that has been in use for quite some time now [3]. Coalescing messages allows users to combine small messages into large ones that effectively send the same amount of data but keep the per message overheads at a minimum. Although programmers can manually coalesce messages to optimize their applications,

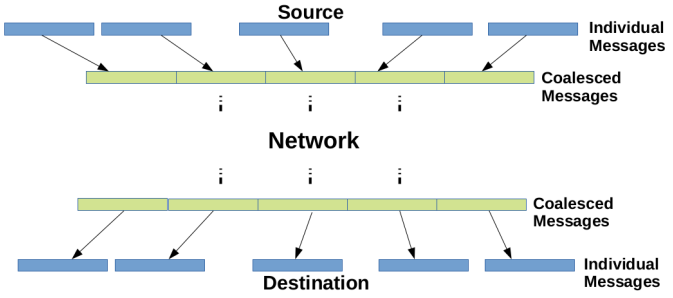

Fig. 1. A diagrammatic representation of message coalescing. Individual active messages are grouped together to form a large message at the sending end which is reconstructed into the original individual entities at the receiving end.

the effort required to correctly achieve this is quite high and is practical only in small and simple applications. Recent work such as Active Pebbles [4] , AM++ [5] and Charm++ [6], have implemented some form of message coalescing solutions provided by runtime systems. Such solutions are largely beneficial in terms of reducing program complexity and coding time. A programmer would simply enable message coalescing and the runtime would intelligently coalesce messages bound to the same destination.

One caveat of this approach is determining how many messages to coalesce in a single message. A single message can be defined by either the size of the buffer, number of messages or a timeout. Since each application has different communication patterns, a single parameter that works best for all conceivable applications simply does not exist. Furthermore, there may be phases in an application where communication is heavy and heavy coalescing would be beneficial, juxtaposed to periods of time when communication is light and sparse that would benefit from different parameters for coalescing. An intelligent adaptive message coalescing approach that dynamically varies its parameters depending upon the application's behavior would be useful.

Recent research [6] has successfully demonstrated a basic adaptive approach for message coalescing where different sets of parameters for coalescing are tried during each iterative step on an all-to-all benchmark using PICS: A Performance-

Analysis-Based Introspective Control System [7]. PICS converged to a decision on coalescing buffer size in 5 decisions. This could be improved upon by creating an advanced adaptive framework able to monitor in real time the network overhead due to fine grained communications. It could select efficient message coalescing parameters based upon that information. Such a system would be able to intelligently vary coalescing parameters based on the phase of the application and would provide a general solution to adaptive message coalescing for applications that do not have a well defined iterative step or a predictable pattern of communication.

HPX [8] is a task based runtime system with real time performance monitoring and tuning capabilities that makes it an obvious choice for experimentation. In order to achieve advanced adaptive message coalescing in HPX, the following steps need to be completed : (i) Implementation of message coalescing in HPX, (ii) Identification of metrics and runtime characteristics that relate to the network overhead associated with fine grained communication, and (iii) Utilization of the identified metrics for adaptive tuning of coalescing for scientific applications. This work presents the first two steps towards the overall goal of achieving automated message coalescing.

As mentioned previously, other task based runtime systems have the ability to coalesce messages. These include Active Pebbles [4],AM++ [5] and Charm++ [6]. Our implementation of message coalescing differs from these implementations in a few fundamental ways. Currently, Active Pebbles, AM++ and Charm++ use buffer size as a means of controlling the granularity of communication. A buffer is allocated and once filled the message is sent. Our approach, however, is to control the number of individual messages to coalesce. Determining when to override messaging coalescing parameters is another important distinction in methodology. While Active Pebbles and AM++ normally send a message when the buffer is full, they also support a flush method which immediately sends the message regardless of the amount of information in it. Charm++ has a periodic check mechanism which performs an immediate send if no messages were sent between subsequent checks. Our implementation of message coalescing allows the coalesced message to be sent after a timeout. When the first message enters the coalescing queue, a timer is set which flushes the coalescing queue on expiration of the timer. Hence, each instance of coalesced messages is sent out either when the coalescing queue is full or when the timeout is triggered. These strategies are necessary to prevent deadlocks caused by messages not being sent due to an insufficient amount of data or an insufficient number of messages waiting to be sent.

The contributions of this research are as follows:

1) Identification of metrics that measure instantaneous network overhead and eventually could be used in adaptive tuning of distributed applications.
2) Implementation of message coalescing in HPX which provides significant improvement in scientific application performance as a result.

II. Implementation Details

A. HPX Runtime System

HPX is a C++ runtime system based on the solid theoretical foundation of the ParalleX [9] model. It exposes a concurrency and parallelism API that is consistent with the current ISO C++ standard. HPX parallel applications can run on both a single machine as well as a cluster with hundreds of thousands of nodes. It is an alternative to the more traditional programming paradigms such as MPI [1]. The architecture of HPX is shown in Figure 2. The HPX threading system employs lightweight tasks, known as HPX threads, that are scheduled on top of operating system threads. A locality in HPX is an abstraction for a physical node. The Active Global Address Space (AGAS) system in HPX provides a mechanism for addressing any HPX object globally. Each object in HPX is assigned a Global Identifier (GID) that is maintained throughout the lifetime of the object even if it is moved between nodes in the system. Local Control Objects (LCOs) are used to synchronize tasks generated by the application. The parcel subsystem is responsible for executing a task remotely and the Performance Counter Framework is used for instrumentation purposes. Additional information on HPX can be found in [8]. What follows is a gentle introduction to the parcel subsystem and the Performance Counter Framework of HPX.

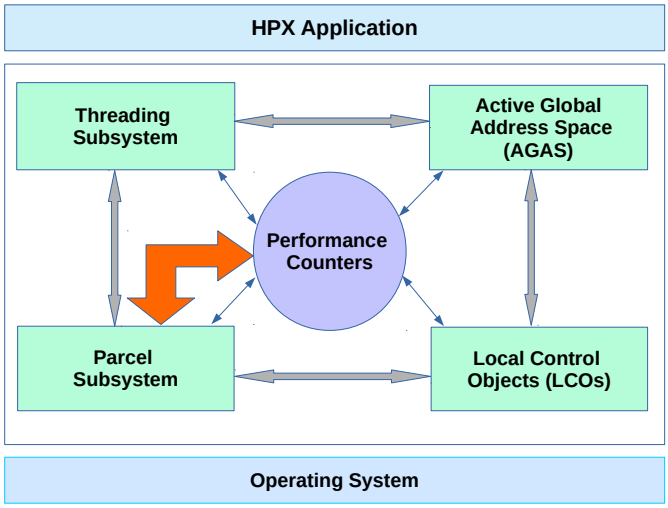

Fig. 2. Architecture of HPX consisting of Threading Subsystem, Active Global Address Space (AGAS), Local Control Objects (LCOs) and the Parcel Subsystem with the Performance Counter Framework interacting with each of the subsystems for instrumentation and debugging purposes.

A parcel(**Par**allel **C**ontrol **El**ement) is a form of an active message [10]. A parcel is created when a method, called ***action*** in HPX terminology, is called remotely. The structure of a HPX Parcel is shown in Figure 3. The ***destination address*** is the location where the method is to be executed, ***action*** is the method to execute, ***arguments*** are the parameters of the method and optionally present ***continuations*** are work that will be executed after the main method in the parcel terminates. In order to transmit a parcel over the network, a

parcel goes through a serialization process converting it into a stream of bytes which is then sent over the wire using existing network protocols. HPX at present supports TCP/IP and is also able to use the MPI communications library for sending sequences of bytes to remote nodes. At the receiving end, a de-serialization process reconstructs the parcel from the received sequence of bytes. The parcel is then converted into a HPX thread and placed in the scheduler queue for execution. The parcel subsystem is responsible for creating the parcels as well as converting a received parcel into a HPX thread.

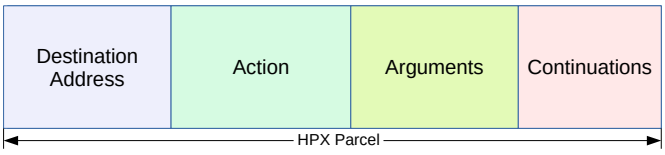

Fig. 3. Structure of a HPX Parcel. A parcel has four components: the destination address (the address of the locality where the parcel is destined); the action, which is the method/function to execute at the destination; the arguments for the function; and optional continuations.

The HPX Performance Counter Framework [11] is used for instrumentation purposes and is able to provide intrinsic information about the state of the application at runtime. It aids in the identification and removal of bottlenecks and assists in debugging. Performance counters let us obtain arbitrary information about the application across the system in a uniform manner. Such information can then be fed into policies for the purpose of runtime adaptivity or can be used for postmortem analysis of the application. Furthermore, HPX allows users to easily create their own performance counters if the existing counters are insufficient.

B. Parcel Coalescing in HPX

The parcel coalescing algorithm is listed in Algorithm 1. Our design revolves around two parameters. First, the length of the parcel queue and second, the wait time. The length of the parcel queue dictates how many parcels are to be coalesced before being sent. The wait time dictates the number of microseconds to wait for the queue to be full before flushing it. Hence, coalesced parcels are sent either when the parcel queue is full or when the wait time expires. Additionally, we employ a limit on the maximum size of the buffer in order to avoid memory overflow errors.

The accuracy of the flush timer is a important factor in the overall working of the parcel coalescing module. The flush timer is designed using Boost's [12] deadline timer that allows the timing mechanism to run in its own dedicated hardware thread. This results in a resolution on the order of microseconds. To verify the accuracy of our flush timer, we ran a series of experiments where a timer was created and set to expire after certain amount of time. We observed that the flush timer fires within on average 33μs of the desired fire time. If we were to use software threads for the purpose of implementing the flush timer, we would have been limited by the time slicing of the Operating System which is in the range of milliseconds.

Algorithm 1 Parcel Coalescing

procedure COALESCING MESSAGE HANDLER
- $nparcels \leftarrow$ *number of parcels to coalesce in a message*
- $interval \leftarrow$ *wait time in microseconds*
- $s \leftarrow$ *state of arriving parcel*
- $tslp \leftarrow$ *time since last parcel*
- **if** $tslp > interval$ **then**
  - **send parcel**
- **switch** s **do**
  - **case** $First$:
    - **Start** *Flush timer*
    - *Queue Parcel*
  - **case** $!First||Last$:
    - *Queue Parcel*
  - **case** $Last(QueueFull)$:
    - **Stop** *Flush timer*
    - *Flush queued parcels*

Another important design consideration when implementing parcel coalescing is when to disable it. The communication pattern of a real life application may change throughout its lifetime. The application may generate large number of parcels at certain points, whereas, there may be points in the application where the number of parcels generated is small. We overcome this hurdle by coalescing the scheduled parcels only when the time between them is less than the maximum wait time. This effectively disables parcel coalescing in cases where parcel generation is sparse. It is important to disable parcel coalescing in cases where parcel generation is sparse because the performance would be negatively impacted as the application will have to wait for the parcel queue to be flushed by the wait timer.

Since parcel coalescing is beneficial in cases where large numbers of parcels are generated, we implemented it in the form of a plug-in rather than incorporating it into the core of HPX. This allows added flexibility for the user as the feature could be easily disabled if needed. Also, parcel coalescing has been implemented on per action basis, and can be easily enabled with minimal change to the existing code by simply adding the macro ***HPX_ACTION_USES_MESSAGE_COALESCING*** as shown in annotation ❶ in Listing 1.

During the course of this study, the following performance counters specific to parcel coalescing were incorporated into HPX:

- ***/coalescing/count/parcels*** that return the number of parcels associated with a particular action,
- ***/coalescing/count/messages*** that return the number of messages generated for a particular action,
- ***/coalescing/count/average-parcels-per-message*** that return the average number of parcels sent in a message for a particular action,
- ***/coalescing/time/average-parcel-arrival*** that return the average time between arriving parcels for a particular

action,
- ***/coalescing/time/parcel-arrival-histogram*** that return a histogram representing the gap between parcel arrival for a particular action.

Performance counters specific to coalescing provide intrinsic information about the application that can be used for debugging and optimization purposes. The performance counters listed above were used for preliminary analysis of parcel coalescing. The above counters also aided in debugging our implementation of parcel coalescing.

III. Network Performance Metrics

This work develops metrics for measuring network overhead of an application. In the context of this work, overhead is defined as the time spent processing information to be communicated across the network. This processing time we call background work. While informative, the time spent processing background work is insufficient to gauge the effects network overhead on the application. An increase in time spent on background work may only indicate a change in application state, eg. communication phase of an application. To understand the influence of overhead, we must look at the ratio of background work to overall execution time. The background work time paired with the overall execution time of the application determines the actual influence of network overheads. The proportion of time spent on overheads to the overall runtime indicates whether significant improvements are possible via a reduction of network overheads.

Parcel coalescing is useful as it reduces the overhead cost per message. In an application that sends millions of messages during its execution, this reduction will be extremely beneficial. After implementing parcel coalescing, we analyzed its effect on the overhead associated with sending and receiving messages. We used two applications, Parquet [13] and a toy application. Details about these applications are provided in section IV. Using the Performance Counter Framework provided by HPX, we obtained intrinsic information about the applications in real time. This section details the metrics we gathered to evaluate the network overheads.

A. Execution Time

We first measured the execution time while varying the number of parcels to coalesce in a single message and the interval to wait before flushing the queued parcels. The size of the problem in each run was kept constant, hence the same number of parcels were generated in each run. The difference between runs was simply the number of messages sent as determined by the coalescing parameters.

B. Task Duration

Next we looked at the overall time spent on executing each HPX-thread or tasks including the overhead. We define task duration using the following equation:

$$t_d = \sum t_{func} \tag{1}$$

where $\sum t_{func}$ is the total time spent by the HPX scheduler executing each HPX thread.

C. Task Overhead

We then looked at the average time spent on thread management for each HPX-thread or tasks. All communication in HPX is done via tasks. Task overhead, is obtained from the */threads/time/average-overhead* performance counter. We calculate task overhead using the following equation:

$$t_o = \frac{\sum t_{func} - \sum t_{exec}}{n_t} \tag{2}$$

where $\sum t_{exec}$ is the time spend by the HPX scheduler doing useful work and $\sum t_{func}$ is the task duration as defined in Equation 1 and n_t is the number of executed HPX threads. We observed a positive correlation between task overhead and overall execution time of our test applications for various coalescing parameters.

D. Background Work Duration

After establishing that task overhead has a positive correlation with the overall execution time, we separated the network related overhead from other overheads. HPX performs network related tasks such as packaging a parcel into a message, serialization, handshaking and locality resolution in the form of background work. We define total time spent doing background work as the background work duration and it is obtained using the following equation:

$$t_{bd} = \sum t_{background-work} \tag{3}$$

Background work duration can be queried using the performance counter */threads/background-work* and was added to HPX as a part of this study.

E. Network Overhead

The network overhead, obtained from the performance counter */threads/background-overhead*, is the ratio of thread background work duration to task duration. Network Overhead is shown in Eq. 4.

$$n_{oh} = \frac{\sum t_{background-work}}{\sum t_{func}} \tag{4}$$

Here, $\sum t_{background-work}$ is the total time spent performing network related work and $\sum t_{func}$ is the total time to reach the completion of each HPX thread. The network overhead performance counter, */threads/background-overhead* was added to HPX as a part of this study.

In subsequent sections, we use the Network Overhead metric defined in Equation 4 in order to measure network overhead of our test applications. Parcel Coalescing is used to demonstrate that the reduction of network overhead increases overall application performance. The control parameters of parcel coalescing can be modified which, in turn, results in a corresponding increase or decrease of network overhead. For network intensive applications, changing network overhead has a demonstrable effect on the application's execution time.

Parcel coalescing attempts to minimize network overhead by sending larger messages across the network via aggregation of smaller parcels into one large message. In the instance of a high volume of parcels in a short window of time this can significantly reduce network overheads.

IV. Experimental Results

A. Experimental Testbed

For our evaluation, we used Marvin thin compute nodes of the ROSTAM [14] cluster located at LSU running the 64 bit Debian GNU/Linux kernel version 3.8.13. Each of the Marvin nodes consists of two Intel Xeon E5-2450 CPUs providing a total of 16 cores, 48GB 1333 MHZ DDR3 memory and 1 TB storage. We used HPX 1.0.0 [15] for the experiments. All applications were compiled using GCC 6.3.0 and Intel MPI 2017.2.174 was used as the underlying MPI implementation.

B. Toy Application

In order to test the effectiveness of parcel coalescing on HPX and its effect on the Network Overhead metric defined in Equation 4, we used a toy application that sends millions of messages containing a single complex double with no direct dependencies between the messages. This example simulates an application where the network overhead is high and is an ideal candidate for testing the effectiveness of parcel coalescing. A condensed version of the code for the toy application is in Listing 1. It shows two nodes sending a million messages to each other and this process is repeated four times. We define the process of sending a million message as a phase as shown in annotation ❷ in Listing 1. We measured the network overhead at specific phases for the toy application using Equation 4. Figure 4 shows network overhead vs. execution time for all sets of parcel coalescing parameters explored in this work. The Pearson's correlation coefficient for our data set was 0.97 which indicates that network overhead and execution time are strongly correlated. We can confidently conclude that larger reported network overhead results in longer execution times.

It is also desirable to see the relationship between parcel coalescing parameters and application runtime. Figure 5 shows the execution time for various values of number of parcels to coalesce in a single messages. The fastest execution time occurs with the largest values of number of parcels to coalesce. This result reflects the toy application's lack of dependency with any other computation or communication. This is further highlighted by our observation that changing the wait time has negligible effect on the execution time. This is due to the fact that the toy application generates the parcels in quick succession such that the parcel queue is almost always filled and the wait time rarely expires.

C. The Parquet Application

One of the applications used in the evaluation was the Parquet [13] application. The self consistent parquet method is a complex physics simulation. The goal is to identify parameters which control the emergence of interesting quantum

```
//Create Action
complex<double> get_cplx()
{
return complex<double>(13.3,-23.8);
}

HPX_PLAIN_ACTION(get_cplx,actn);
HPX_ACTION_USES_MESSAGE_COALESCING(actn); ❶

//Create instance of the actions
actn act;

vector<hpx::future<complex<double>>> vec;
vec.reserve(numparcels);

//Find the other locality
auto localities=hpx::find_remote_localities();
auto other=localities[0];

int num_repeats=4;
//Repeat num_repeats times
for (int j = 0; j < num_repeats; j++)
{
for (int i = 0; i < numparcels; ++i)
{
vec.push_back(hpx::async(act, other));
}
//Wait for all the tasks to complete
hpx::wait_all(vec);
}
```

❷

Listing 1. Artificial example application used to generate and send parcels from one node to another.

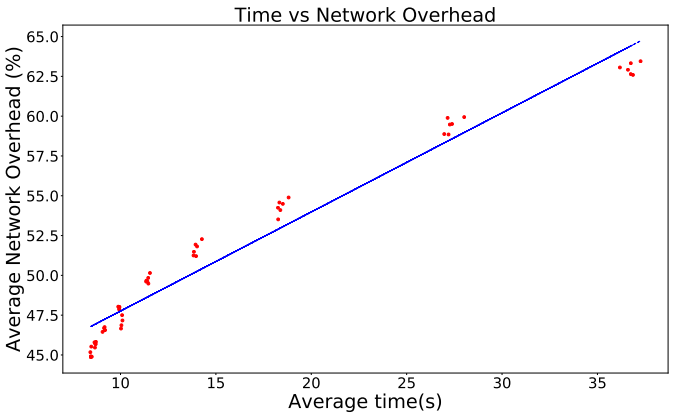

Fig. 4. Scatter Plot of the average network overhead per phase vs average execution time per phase for the toy application. Each dot represents a set of parcel coalescing parameters. Average overhead is the average for four phases. As the network overhead decreases, the execution time also decreases. A Pearson's correlation coefficient of 0.97 indicates a strong positive correlation between network overhead and runtime.

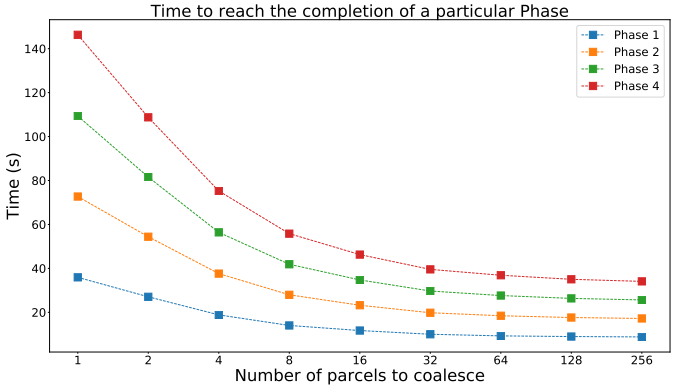

Fig. 5. Time to reach the completion of a particular phase in the toy application for various values of number of parcels to coalesce in a single message with a wait time of 4000μs. In this example, as more parcels are coalesced, the time to reach the completion of a phase decreases.

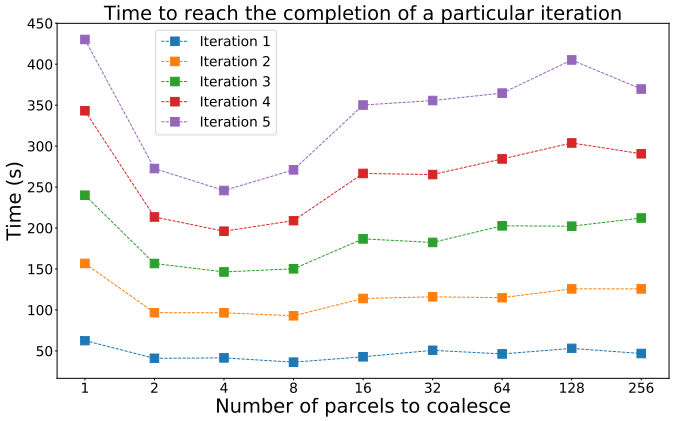

Fig. 6. Time to reach the completion of different iterations in the parquet application for various numbers of parcels coalesced in a single message with a wait time of 4000μs. Each color indicates a different iteration. There is a clear decrease in overall runtime from coalescing one parcel in a message to coalescing two. The minimum runtime is found when coalescing four parcels in a message after which the runtime increases due to a suboptimal choice of parcel coalescing parameters.

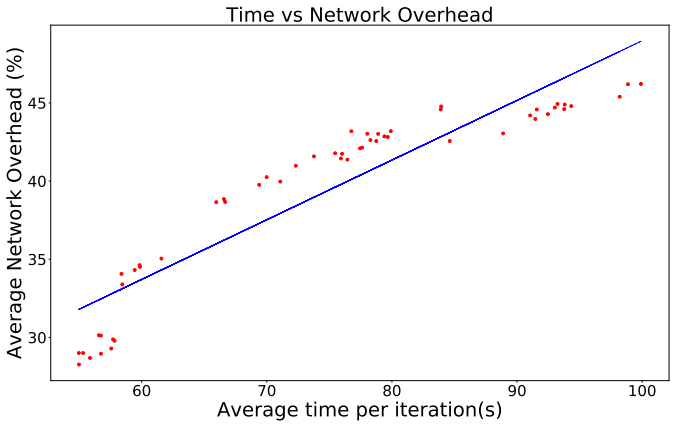

Fig. 7. Scatter Plot of Average Network Overhead Vs Average time per iteration for the Parquet Application. Each dot represents a set of parcel coalescing parameters. A Pearson's correlation coefficient of 0.92 was calculated indicating a strong positive correlation between network overhead and runtime for the parquet application.

phenomena in strongly correlated materials. The simulation requires the use of many rank-3 tensors composed of complex doubles. The linear dimension (N_c) of the simulation controls the tensor size. The tensor contains N_c^3 complex doubles. The memory required by the simulation can approach terabytes. Accommodating such large quantities of data requires the simulation to be executed on multiple nodes. Throughout the simulation all the data from each node must be broadcast to the other nodes. The rotation phase sends $8*N_c^2$ parcels containing N_c elements. No message depends on another and they can be sent in parallel.

For the trial simulation executed on four nodes, N_c = 512 was chosen as it uses a non trivial amount of memory and it exposes high network utilization. Tests indicate the timing of an individual run will not be likely to vary much from the averages reported. In our experiment we coalesced 4 parcels into a single message and waited 5000μs before flushing the parcel queue. We ran the experiment 100 times. The calculated Relative Standard Deviation of the trial was less than five percent which indicates that the random fluctuations of an individual run should not influence the trends reported.

Measurements of network overhead and total execution time demonstrate the importance of parcel coalescing in a communication heavy problem. To account for the random nature of any application that involves heavy network traffic, the application was run three times for each set of parameters. The following results show the averages of the measured values from the three independent runs. Figure 6 shows the variation in overall time to complete different iterations of the parquet application coalescing different numbers of parcels in a single message with a wait time of 4000μs. We observed a clear decrease in runtime by coalescing two parcels in a message. Increasing the number of parcels to coalesce improved the runtime further. It was observed that coalescing four parcels in a message resulted in the minimum time. Further increasing the number of parcels in a message adversely affects the runtime. These trends are more pronounced in the later iterations due to cumulative effects.

In order to further understand the relationship between parcel coalescing parameters and the overall runtime of the parquet application, we performed a parameter sweep running the parquet application with increasing the value of the parcel coalescing parameters until the execution time showed a clearly increasing trend. As seen in Figure 8, bands along the axes where number of parcels to coalesce in a single message is one or when wait time is set at 1μs highlight the largest runtimes. This choice of parameters effectively disable parcel coalescing thus resulting in the large runtime seen. We get an immediate reduction in runtime with two parcels to coalesce in a single message. The maximum reduction in runtime was seen with coalescing four parcels in a message and wait time of 5000μs.

For the same runs, performance counters reported the net-

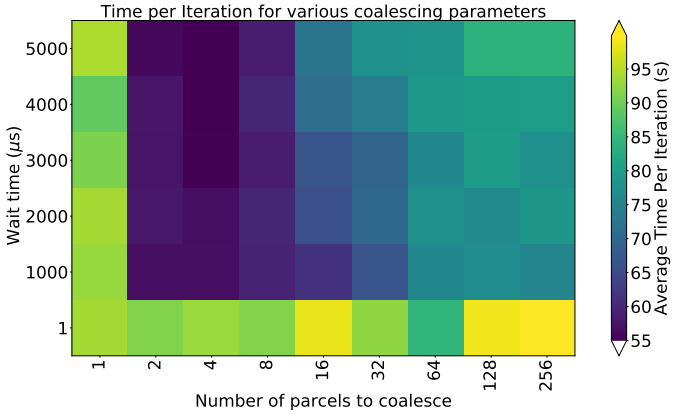

Fig. 8. Average time per iteration for different numbers of parcels to coalesce into a single message and increasing wait times before flushing the parcel queue. Parcel coalescing is effectively disabled when a message contains only a single parcel or only 1 μs wait time before flushing the parcel queue. This produces slower execution times as seen in the bars along the horizontal and vertical axes.

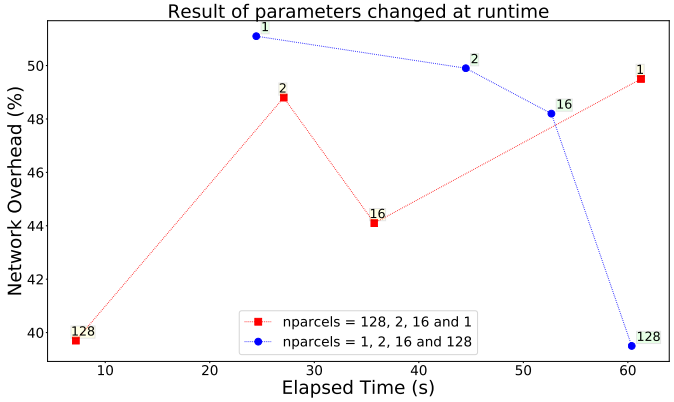

Fig. 9. Network overhead for various values of number of parcels to coalesce in a single message each phase with a wait time of 2000μs for two different runs of the toy application. One run starts with optimal value 128 parcels to coalesce in a single message producing the lowest overhead that increases substantially in the subsequent phases with suboptimal parameter choice. Another run starts with a single parcel every message that produces the highest overhead that substantially decreases with favorable parameter selection.

work overhead as defined in Eq. 4. Figure 7 graphs various values of coalescing parameters against the value of the average network overhead counter. It was seen that the parcel coalescing parameter that resulted in lower overhead additionally had lower execution time. Our calculated Pearson's correlation coefficient of 0.92 indicated a strong positive correlation. Most of the parameter sweep results in larger overheads than the optimum parameter. This implies that an arbitrary choice of parcel coalescing parameters will likely result in suboptimal performance. The choice of parcel coalescing parameters must therefore be done carefully.

D. Instantaneous Measurements

In order to show that the metrics introduced in section III can be used to signal networking efficiency, we reran our artificial application from Listing 1 with various settings for number of parcels to coalesce with a wait time of 2000μs. Here, instead of using the same number of parcels to coalesce throughout the lifetime of the application, we changed the number of parcels to coalesce into a single message during different application phases. As seen in Figure 9, we start with 128 parcels to coalesce in a single message which produces the lowest network overhead. Upon changing to suboptimal values in subsequent phases, we see that network overhead increases along with the total time to complete the phase. Furthermore, a different run of the same application which started with a suboptimal value of coalescing, one parcel per message, produced the highest overhead in the beginning but changed when a favorable number of parcels were coalesced which reduced the network overhead in subsequent phases. We also observe that suboptimal values of coalescing parameters produce larger phase completion time. This preliminary result indicates that our metric can be used to signal when changes in coalescing parameters would be beneficial in an adaptive setting.

V. Discussion

The performance of applications that generate large numbers of small parcels in task based runtime systems such as HPX can be improved by coalescing these small parcels into larger ones. This improvement is largely due to reduction in network overheads as fewer messages are created. However, static approaches towards coalescing parameter selection can only provide limited gains in performance. Adaptive techniques are needed to make further reductions in application execution times. Charm++ [6] has shown the effectiveness of automatic configuration parameter selection using PICS: A Performance-Analysis-Based Introspective Control System [7]. Their approach tested a set of configuration parameters for each iteration of the application and chose new parameters based on the performance measured during that iteration. This approach to adaptive tuning is only suited for iterative applications, and therefore, this technique is unable to consider the phase of the application. The methodology introduced in this research improves upon the state of the art by introducing new intrinsic performance counters which provide the current state of the application in real time. Using information obtained from such counters, one can make a distinction between different communication phases of the application and select configuration parameters accordingly. Metrics identified in this research have shown a strong correlation with execution time of the test applications. Furthermore, Figure 9 demonstrated that changing the coalescing parameters at runtime could influence the instantaneous network overhead. This result indicates that data obtained from intrinsic performance counters could be used to make adaptive decisions. This allows for intelligent adaptive behavior where one can employ different configuration settings depending upon the phase of the application.

VI. Conclusions and Future Work

In this paper, we implemented parcel coalescing in HPX as a means to reduce the cost of overhead associated with sending and receiving messages. Our implementation of parcel coalescing in HPX provided marked reduction in total runtime for the toy application as well as a real physics simulation, Parquet. This paper also presented methods to measure the network overhead within task based runtime systems. We were able to establish strong positive correlation between the network overhead measured using our metric and the overall runtime of both applications. We showed that the benefits from parcel coalescing are due to the reduction in overheads associated with message transmission. As seen in the two applications presented in this research, choosing the best coalescing parameters is important as execution time increases with suboptimal parameter selection. Additionally, the large difference between parameters that produce the shortest time for the toy application and the parquet application highlight the need to identify the best set of parameters for each application. We also demonstrated that alternatives to a brute force method of determining these parameters is required for reasonable application productivity.

The metrics identified in this research aid in evaluating network efficiency by giving us an intrinsic view of the underlying network overhead which would be difficult to measure using conventional methods. Our research gives the user an ability to assess performance from a perspective other than that of execution time. This allows a user to analyze an application in real time and observe the effect of varying parcel coalescing parameters on network overheads at runtime. The strong positive correlation between execution time and network overhead opens new possibilities for advanced adaptive solutions for parcel coalescing. The runtime system could tune its parcel coalescing parameters dynamically by evaluating the overhead counters provided by the Performance Counter Framework. In the future, metrics and techniques defined in this research could be used as a basis for the adaptive tuning of a broad set of messaging parameters.

VII. Acknowledgments

The authors would like to thank Mark Jarrell, Juana Moreno and Ka-Ming Tam for their valuable input and suggestions throughout the study. This work was partly funded by the NSF EPSCoR LA-SiGMA project under award #EPS-1003897 ,the NSF STORM project under the award #ACI-1339782 and NSF Phylanx project award #1737785. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] M. P. Forum, "Mpi: A message-passing interface standard," Knoxville, TN, USA, Tech. Rep., 1994.

[2] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson, "Effects of communication latency, overhead, and bandwidth in a cluster architecture," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: ACM, 1997, pp. 85–97. [Online]. Available: http://doi.acm.org/10.1145/264107.264146

[3] C. D. Pham, "Comparison of message aggregation strategies for parallel simulations on a high performance cluster," in *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (Cat. No.PR00728)*, 2000, pp. 358–365.

[4] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Active pebbles: Parallel programming for data-driven applications," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011, pp. 235–244. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995934

[5] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Am++: A generalized active message framework," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2010, pp. 401–410.

[6] L. Wesolowski, R. Venkataraman, A. Gupta, J. S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. R. Quinn, and L. V. Kale, "Tram: Optimizing fine-grained communication with topological routing and aggregation of messages," in *2014 43rd International Conference on Parallel Processing*, Sept 2014, pp. 211–220.

[7] Y. Sun, J. Lifflander, and L. V. Kalé, "Pics: A performance-analysis-based introspective control system to steer parallel applications," in *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, ser. ROSS '14. New York, NY, USA: ACM, 2014, pp. 5:1–5:8. [Online]. Available: http://doi.acm.org/10.1145/2612262.2612266

[8] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: ACM, 2014, pp. 6:1–6:11. [Online]. Available: http://doi.acm.org/10.1145/2676870.2676883

[9] H. Kaiser, M. Brodowicz, and T. Sterling, "Parallex an advanced parallel execution model for scaling-impaired applications," in *Proceedings of the 2009 International Conference on Parallel Processing Workshops*, ser. ICPPW '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 394–401. [Online]. Available: http://dx.doi.org/10.1109/ICPPW.2009.14

[10] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: A mechanism for integrated communication and computation," *SIGARCH Comput. Archit. News*, vol. 20, no. 2, pp. 256–266, Apr. 1992. [Online]. Available: http://doi.acm.org/10.1145/146628.140382

[11] P. Grubel, H. Kaiser, J. Cook, and A. Serio, "The performance implication of task size for applications on the hpx runtime system," in *Proceedings of the 2015 IEEE International Conference on Cluster Computing*, ser. CLUSTER '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 682–689. [Online]. Available: http://dx.doi.org/10.1109/CLUSTER.2015.119

[12] Boost, "Boost C++ Libraries," http://www.boost.org/, 2017.

[13] S. X. Yang, H. Fotso, J. Liu, T. A. Maier, K. Tomko, E. F. D'Azevedo, R. T. Scalettar, T. Pruschke, and M. Jarrell, "Parquet approximation for the 4x4 Hubbard cluster," vol. 80, p. 046706, 2009.

[14] S. Group, "Running HPX on ROSTAM," https://github.com/STEllAR-GROUP/hpx/wiki/Running-HPX-on-Rostam, 2017.

[15] H. Kaiser, B. A. L. aka wash, T. Heller, A. Berg, J. Biddiscombe, A. Bikineev, G. Mercer, A. Schfer, atrantan, A. Serio, J. Habraken, M. Anderson, S. R. Brandt, M. Stumpf, D. Bourgeois, M. Copik, K. Huck, V. Amatya, L. Viklund, Z. Khatami, D. Bacharwar, S. Yang, E. Schnetter, Bcorde5, M. Brodowicz, L. Troska, B. Wagle, S. Upadhyay, Z. Byerly, and H. Brakmi, "STEllAR-GROUP/hpx: HPX V1.0: The C++ Standards Library for Parallelism and Concurrency," Apr. 2017. [Online]. Available: https://doi.org/10.5281/zenodo.556772