An MPI-Stream Hybrid Programming Model for Computational Clusters

Emilio P. Mancini, Gregory Marsh, and Dhabaleswar K. Panda
Department of Computer Science and Engineering,
The Ohio State University
{mancini, marshgr, panda}@cse.ohio-state.edu

***Abstract*—The MPI programming model hides network type and topology from developers, but also allows them to seamlessly distribute a computational job across multiple cores in both an intra and inter node fashion. This provides for high locality performance when the cores are either on the same node or on nodes closely connected by the same network type. The streaming model splits a computational job into a linear chain of decoupled units. This decoupling allows the placement of job units on optimal nodes according to network topology. Furthermore, the links between these units can be of varying protocols when the application is distributed across a heterogeneous network.**

In this paper we study how to integrate the MPI and Stream programming models in order to exploit network locality and topology. We present a hybrid MPI-Stream framework that aims to take advantage of each model's strengths. We test our framework with a financial application. This application simulates an electronic market for a single financial instrument. A stream of buy and sell orders is fed into a price matching engine. The matching engine creates a stream of order confirmations, trade confirmations, and quotes based on its attempts to match buyers with sellers. Our results show that the hybrid MPI-Stream framwork can deliver a 58% performance improvement at certain order transmission rates. [1]

I. Introduction

Achieving optimal performance in a parallel application is a hard task when underlying topology becomes complex, and the network parameters can affect overall performance in unpredictable ways. The increased number of nodes in modern computational systems introduces implicit heterogeneity, for example, when using switch hierarchies. In these cases, some links may have higher latency and lower bandwidth, because the data crosses a different number of switches. MPI implementations are the most common frameworks used for parallel programming, but they hide the system topology [1]. Therefore, exploiting the locality can improve the performance of a parallel MPI program.

In some cases, users want to exploit clusters of clusters, so that one MPI program can spawn across multiple clusters. Often this architecture uses a high performance network for intra-cluster communication, and low performance communication for inter-cluster data transfers. A flat model such as MPI with heavy coupled tasks can only exploit such configurations with difficulty. Stream programming models, instead, use independent computational units, that can easily exploit heterogeneous networks. However most of the algorithms commonly used in parallel programming need to be reformulated for stream use [2].

In this paper, we propose a hybrid approach that combines the Stream model to build an application across heterogeneous communication systems, and an MPI model to exploit heavy local computation on computational clusters. We designed a framework with a minimal set of primitives to support distributed hybrid Stream/MPI programming.

One of the most studied areas of application of the stream paradigm is the financial area. We tested the proposed framework with a financial application comparing two models: the first with a pure MPI model, the second with a hybrid Stream/MPI paradigm. The results show that the lighter overhead and data structures in the stream portion of the hybrid model deliver a 58% performance improvement in some scenarios as compared to the pure MPI model.

In this paper, in Section II we present the stream and MPI programming models. Then in Section III we propose a way to mix the two models. In Section IV we introduce a new streaming framework, and in Section V we apply it to a financial application in order to compare a pure MPI implementation with a hybrid MPI-Stream architecture. In Section VI we present a performance evaluation with the application. After a review of related work, in Section VIII we draw conclusions and outline future work.

II. The stream and MPI programming models

The Message Passing Interface (MPI) is a specification that describes a parallel programming model. MPI is the de-facto standard in parallel programming, providing a well defined interface and several efficient implementations. It explains how several processes can move data between their address spaces using communication operations, such as, point-to-point, collective, or one-sided. Besides this, MPI offers other interesting features like Dynamic Process Management, parallel IO, and new features are continuously proposed [1].

Every MPI program is a set of autonomous processes that do not run necessarily the same code; they elaborate and exchange data, through message passing, following the Multiple Instructions Multiple Data (MIMD) model. The basic operations are point-to-point transactions that involve two processes: one that initiates the data transfer with *send* routines, and the other that is ready to receive incoming data with a *receive* function. Collective and one-side operations can optimize such processes in various scenarios. MPI tries to hide the underlying network topology presenting a flat view to the processes. While the user's program does not know about the hardware structure, several MPI implementations try to

[1]This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CNS-0403342, #CCF-0702675, #CCF-0833169, #CCF-0916302 and #OCI-0926691; grant from Wright Center for Innovation #WCI04-010-OSU-0; grants from Intel, Mellanox, Cisco, QLogic, and Sun Microsystems; Equipment donations from Intel, Mellanox, AMD, Advanced Clustering, Appro, QLogic, and Sun Microsystems.

exploit any knowledge they have to optimize the most complex operations such as the collectives [3].

The Stream approach is quite different. A parallel application using the Stream Programming Model is composed of multiple tasks, called "kernels", connected by point-to-point links. These connect two successive kernels, so that each one sends its output as input to another one. The next kernel gathers and elaborates the data before placing it in another stream. Therefore, the streams are an unbounded flow of data records that link computational units. Each kernel is independent; the only way it has to communicate is through streams. It does not have control on arriving data; it can only filter it and then chose to discard or send forward the results of its computations [2], [4].

Fig. 1 shows a generic acyclic communication stream model. The basic way to compose the kernels is the pipeline, but more advanced schemas can use *splits*, *joins* and *feedbacks*. The splits can duplicate a stream, so that successive kernels gather the same data set, or it can divide the records so the successive kernels take only a part of original data. In other words, data streams can be duplicated on two physical links, or separated so that the computational effort can be divided on two different kernels. The join operations put different streams in the same receive queue.

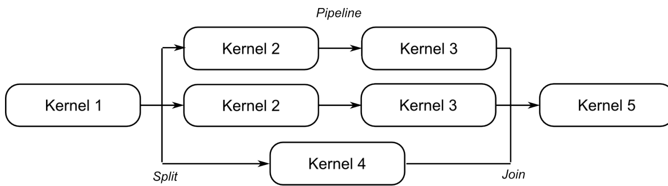

Fig. 1. Generic streams model

III. Designing a Hybrid Stream/MPI Model

As described in the preceding section, MPI implementations are the ideal way to exploit strictly interconnected nodes, and have high performance on computational clusters. But when the topology becomes more complicated, it is harder to exploit it in an optimal way [5].

MPI uses a flat view of its parallel virtual machine, so, for example, the joint use of different clusters, and the access to internal hidden nodes can be a hard and time-consuming task. The resulting performance may not be so good, because the presence of slow links, can reduce the overall speed due to synchronization and collective operations involving both, fast and slow links.

On other side, streams can easily exploit heterogeneous networks. Because the kernels are highly independent, the user can design the kernel chain so that it can exploit faster machines to handle complex duties. The absence of explicit synchronizations and the decoupling of computational units can result in better performance when using heterogeneous networks. If some part of the computation is particularly heavy, the user can assign it to a parallel MPI application, rather than to a single sequential kernel as in Fig. 2. In this way the latency of the whole chain should be lower.

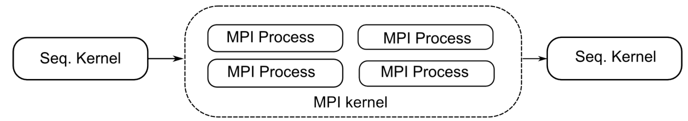

Fig. 2. The use of an MPI application as stream kernel.

When heterogeneous technologies are involved, the launching process is complex. It requires a description of the whole system, and running and synchronization of both MPI and sequential programs on different nodes. As described in Fig. 3, the user runs a first level launcher, which launches sequential tasks as new remote processes, and parallel kernels using the standard MPI launchers. An XML file can provide all necessary information to dynamically produce MPI hostfiles and startup scripts. When it starts, every kernel has to call an initialization routine, in order to build a local graph data structure that the middleware can use to direct the communication; this graph is similar to the one shown in Fig. 1.

Every kernel, at this point, should register itself with the middleware, so that it can run the right function on the right machine. A different approach is to let each kernel poll the stream autonomously without explicit middleware management. We used the first method for sequential kernels, and the second for MPI kernels. When the application registers the kernels, it can specify some additional information about their behavior, for example if they are stateless or statefull, or if they use a polling or callback model to elaborate streams. At this point, the application can ask the middleware to start the kernel processes, and open the streams with the Get and Put operations. A fragment of a typical program in the model we propose is:

```
/* Source kernel: opens an output
   stream */
osf_Result_t SourceKernel(
         osf_KernelContext_t *ctx) {
  static osf_Stream_t *s = NULL;
  if (s==NULL)
      osf_Open(&s, OSF_STR_OUT, 1);
  ...
  record = sin(t)+sin(4+2*t);
  osf_Put(s, &record, sizeof(record) );
  return OSF_ERR_SUCCESS;
}

/* Destination kernel: takes the
   data from source kernel and
   elaborates */
osf_Result_t FilterKernel(
     osf_KernelContext_t *ctx) {
  static osf_Stream_t *sIn = NULL;
  if (sIn==NULL)
      osf_Open(&sIn, OSF_STR_INPUT, 0);
```

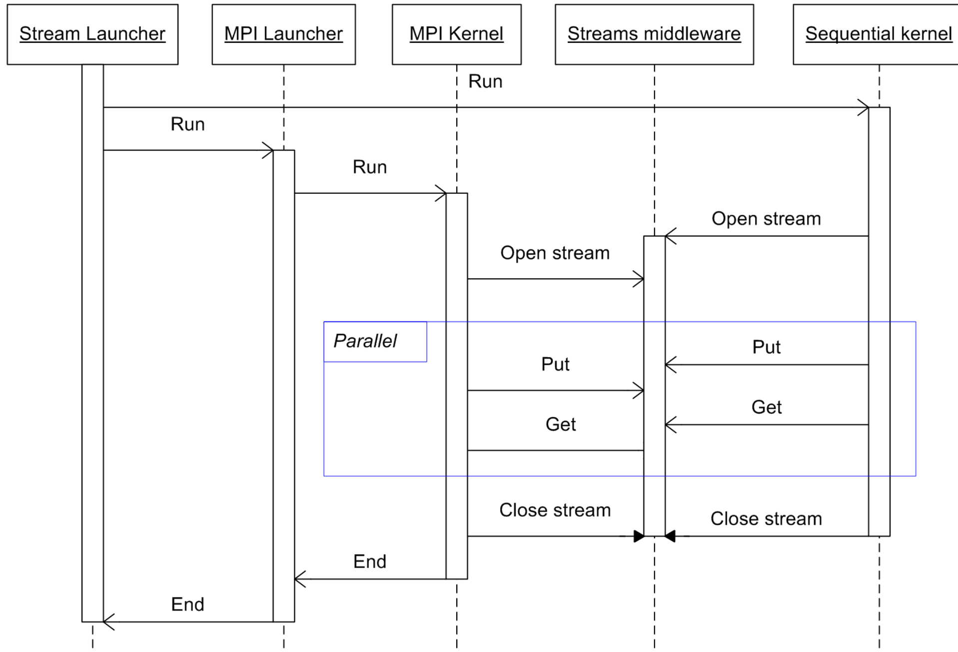

Fig. 3. The sequence diagram of the hybrid MPI/Stream model.

```
  ...
  res = osf_Get( sIn, &x, sizeof(double),
         &receiv );
  return OSF_ERR_SUCCESS;
}

/* Main function: registers
   the kernels and starts the
   streams */
int main (...) {
  osf_KernelContext_t *kctx;
  osf_Init(...);
  osf_RegisterKernel(0,
    OSF_KRNTYP_POLLING, OSF_KRN_STATELESS,
    SourceKernel, &kctx);
  osf_RegisterKernel(1,
    OSF_KRNTYP_POLLING, OSF_KRN_STATELESS,
    FilterKernel, &kctx);
  osf_StartStreams();
  osf_Finalize();
}
```

In the preceding example, each kernel is coded in a function with a well-defined interface. We have chosen a functional implementation model to be consistent with the MPI functional logic. Other solutions require the use of compiler or language extensions [6].

The main procedure registers the kernel procedures using the *osf_RegisterKernel* API. In this way the middleware knows what are the kernels and how to associate each routine to the internal graph. Then it starts the streams (*osf_StartStreams()*) according to the external launcher indications, activating one or more kernels on each computational node.

IV. The streaming framework

To implement a framework supporting the hybrid Stream/MPI approach we studied the architecture shown in Fig. 4. The core module is a modularized communication subsystem. The application can interact with the system using the Abstract Communication Interface. The framework hides the network type so that different segments of the stream chain can use different communication technologies. When the user starts more than one kernel on the same node, the middleware uses Posix threads and IPC in order to optimize local communication. When the launcher starts the kernels on different nodes it can chose the network technology as a function of its hardware and of its needs. For example, it can reserve high performance networks for communication intensive MPI kernels, and low performance networks to connect sequential kernels. Besides supporting different network technologies, the current prototype uses only socket communication. In the future, we plan to extend the communication modules.

One of the goals of this study is to identify a minimal set of primitives for designing a framework supporting the integration of stream and MPI programming models. Table I presents this set, leaving out utility functions like initialization routines. The *osf_RegisterKernel()* function inserts information about each kernel into the framework. The framework gathers such information from two sources: the compiled program, using the API and an external configuration file, which maps the kernels with the computational nodes. After the program registers the kernels, it can start them as new processes using *osf_StartKernels()*.

To communicate, a kernel needs to open a stream with the

Operations	Description
RegisterKernel()	Register the kernel in the middleware;
StartKernels()	Optional hints (for future purposes);
Open()	Open or subscribe a stream;
Put()	Put a set of record in the stream;
Get()	Get records from incoming stream.

TABLE I
THE PROPOSED MINIMAL SET OF PRIMITIVES.

osf_Open() function. This creates the required connections to the right kernels so that the sender can transmit its data with the *osf_Put()* operation, and the destination can gather data with the *osf_Get()* function.

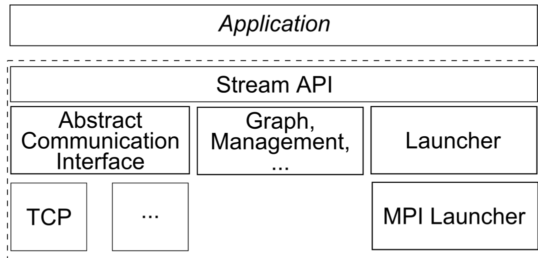

Fig. 4. The framework prototype high level architecture.

V. CASE STUDY WITH A FINANCIAL APPLICATION

We tested our framework on a simulation of the market for a single financial instrument such as a stock. The code was originally developed by the ZeroMQ project as a test application for its messaging middlware [7]. Our adaptations (Figures 5 and 6) have 4 major components: a Random Data Generator, a Matching Engine, an Order Confirmation receiver, and a Store. The Matching Engine is the core of the simulation and contains data structures that receive, hold, and process orders from the Random Data Generator. The order type, price, and volume are randomly generated numbers, the ranges of which were choosen by the simulation's original designers. The order type may be either a bid (buy) or ask (sell). The price of an order is an integer between 450 and 549 inclusive. The volume is the amount to be bought or sold and is an integer between 1 and 100 inclusive. If a bid price is equal to or greater than the lowest available ask price, the Matching Engine executes a trade and sends a confirmation and a quote with the current market price to the Order Confirmation receiver. The Order Confirmation component sends everything it receives to the Store which archives all messages on disk.

1) Data Structures and Global Variables: The Matching Engine simulates a market with the help of an OrderBook vector comprised of double-ended queues and two global variable integers (Figure 7).

vector <double-ended queue> OrderBook[0..999]: Each index of OrderBook vector represents a price. Orders with the same price are stored in the double-ended queue at the vector element for their price-index. These queues preserve

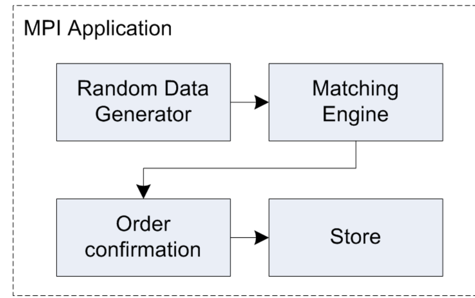

Fig. 5. The case study financial application chain: pure MPI implementation.

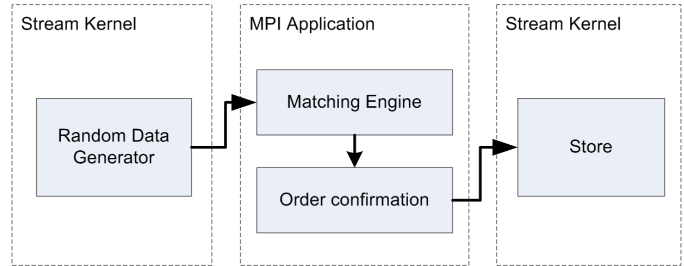

Fig. 6. The case study financial application chain: hybrid implementation.

orders that cannot be traded immediately while ensuring that any order may be matched for a trade in the sequence that it was received at the Matching Engine.

integer min_ask: The OrderBook index storing ask orders with the lowest available selling price. Any orders queued in OrderBook elements at, and to the right of, min_ask are ask orders.

integer max_bid: The OrderBook index storing bid orders with the highest available buying price. Any orders queued in OrderBook elements at, and to left of, max_bid are bid orders.

2) Matching Engine Algorithm: The Matchine Engine is composed of a recieve loop that retrieves orders sent over the network by the Random Data Generator and two routines that simulate the market by processing either bid or ask orders, depending on order type.

a) Matching Engine Receive Loop:

loop
  Recieve $order$
  if $order.type = ask$ **then**
    Ask_Order_Routine($order$)
  else
    Bid_Order_Routine($order$)
  end if
end loop

b) Overview of Order Processing Routines: In brief the Ask_Order_Routine and the Bid_Order_Routine each do the following. More precise details are presented in the Appendix at the end of this paper.

1) Compare the price of the received order against min_ask or max_bid to determine if the order can

be immediately matched for a trade with orders already enqueued in the OrderBook (Figure 7).

2) If the received order can not be matched immediately then it is enqueued in the OrderBook at its price index.
3) When executing trades, each routine considers the received order's volume. If there is not enough matching volume at one price level, each routine will match whatever volume is available and then search other matchable price levels in the OrderBook in order to totally satisfy the remaining volume in the received order. For example an ask order of price 500 with volume 100 is received and the max_bid is 510 while the min_ask is 500. The Ask_Order_Routine first tries to dequeue as many bid orders as needed at price 510 to satisfy the ask order's volume of 100. Suppose that the bids enqueued at 510 only total 30 in volume. Trade confirmations are sent for these matched trades. The routine will then decrement max_bid and try to find matching bids in the price range 509 to 501, creating as many trades as needed to fill the remaining ask order volume of 70. If the ask order can only be partially filled the volume of the order is adjusted and the order is enqueued in the OrderBook for future matching of the remaining volume.

Each routine runs in $O(n * max_price_depth)$ where n is the range of prices (450...549 in this simlation) and max_price_depth is the size of the largest doubled-ended order queue within the OrderBook. However in this simulation as orders rapidly accumlate across the range of prices, the runtime of these routines approach $O(1 * max_price_depth)$ as more often that not a price match can be immediately found without searching throughout the price range for compatible orders.

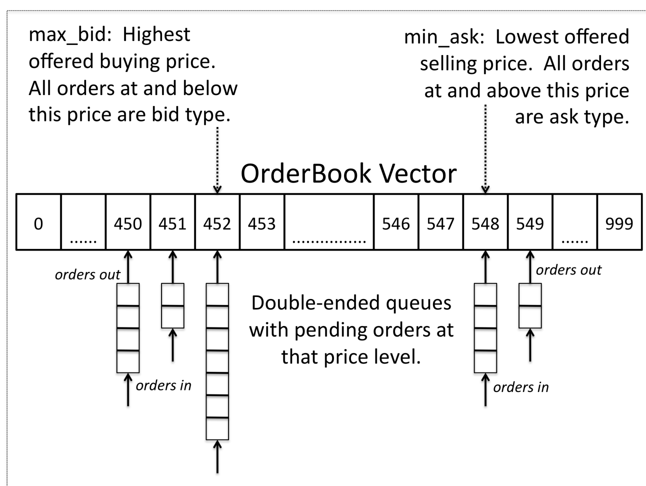

Fig. 7. Matching Engine Data Structures.

VI. Performance Evaluation

A. Experimental Setup

As detailed above, the Matching Engine receives orders and matches the prices in buy orders to the prices in sell orders. A compatible price match results in a trade. The Matching Engine produces three types of messages which are sent back to the Order Confirmation component: trade confirmations, order confirmations, and price quotes. Order confirmations are produced when there is no matching price to make a trade. Price quotes occur when the stock's maximum bid or ask price changes due to trading activity. During the course of their operation, the Random Data Generator, Matching Engine, Order Confirmation component, and Store produce performance measurements. These measurements are taken with a sampling interval, which we set for every 20,000 orders. The Random Data Generator generates the following measurement:

1) Orders per second sent from the Random Data Generator to the Matching Engine.

The Order Confirmation component measures the following:

1) Order Completion Time: Time in seconds between when an order is sent by the Random Data Generator and when either a trade confirmation or order confirmation is received by the Order Confirmation component.
2) Sum of trade confirmations, order confirmations, and quotes received per second from Matching Engine by the Order Confirmation component.

The Matching Engine produces the following measurements:

1) Orders per second received from the Data Generator.
2) Sum of trade confirmations, order confirmations, and quotes sent per second from the Matching Engine to the Order Confirmation.

Our evaluation uses a cluster consisting of Intel Xeon Quad dual-core processor hosts. Each host node has 6GB RAM and is equipped with a 1 Gigabit Ethernet network interface controller. The operating system for each node is Red Hat Enterprise Linux Server 5. For MPI operations we used MPICH2 Version 1.2 middleware [8]. Each message type (orders, confirmations, and quotes) exchanged between the components has a different sized payload. We counted the number of each message type created during a typical run and calculated a weighted average message size of around 9 bytes. The application's performance may be tested by varying the number of random orders per second created by the Random Data Generator as well as varying the programming model scheme (MPI or Hybrid MPI-Stream). In our tests we used order generation rates of 5,000, 10,000, 15,000, and 20,000 orders/second and created 10,000,000 orders at each rate. We recorded the resulting measurements at each rate. Beyond 20,000 orders/second, the application stressed the limits of the hardware and would not record a full set of measurements.

B. Experimental Results

Figures 8 and 9 show our experimental results. Figure 8 summarizes measurement 1 on the Order Confirmation component. The MPI-Stream Hybrid scheme on Gigabit Ethernet (hygrid-ge) achieves a faster order completion time than the corresponding MPI-only scheme (mpi-ge) at order generations rates of 5,000 and 10,000 orders/sec. We feel that this result

reflects the lighter overhead and minimal data structures of the streaming framework within the Hybrid scheme, as compared the MPI scheme where all processes must make use of the MPICH2 middleware. At rates of 15,000 orders/sec and above, performance for the Hybrid scheme begins to deteriate. The MPI-only scheme does not degrade until 20,000 orders/sec and performs better than Hybrid at 15,000 orders/sec. We feel this reflects a “sweet spot” where the high overhead of its middleware is efficiently amortized over a high network load that the middleware effectively bears. We are carrying out further optimizations in the hybrid design to obtain better performance.

Figure 9 summarizes measurements 1 and 2 from both the Data Generator and Matching Engine, as well as measurement 2 from the Order Confirmation component. The lower line shows that measurement 1 for the Data Generator and Matching Engine was the same: the Matching Engine was able to read orders from the network at the same rate as the Generator was able to produce them. The upper line shows measurement 2 for the Matching Engine and Order Confirmation component is the same for both the MPI and Hybrid schemes except at 20,000 orders per second. At all but this order generation rate, the Order Confirmation component was able to read confirmations and quotes from the network at the same rate as the Matching Engine was able to generate them. Although the MPI-only scheme also shows a high order completion rate at 20,000 orders/sec, the more complex structures in the MPICH2 middleware allow for a high sustained reading of data buffers from the network at this traffic level.

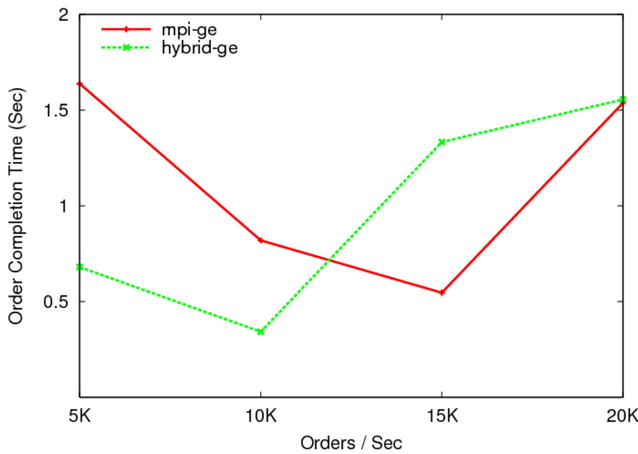

Fig. 8. Financial Application Order Completion Time

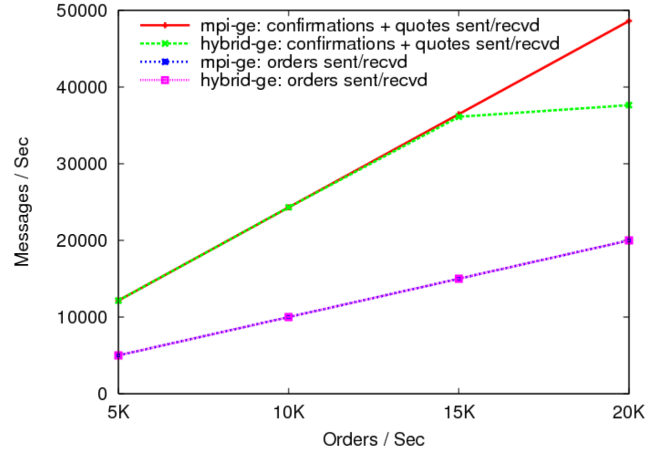

Fig. 9. Financial Application Message Rate

VII. Related work

Several studies try to apply the MPI model to heterogeneous scenarios, exploiting for instance clusters of clusters. Some implementations, like MPICH-G2 uses the Grid technologies to hide the heterogeneity and to allow users to manage it. MPICH-G2 uses the Globus Toolkit services to implement communication and management layers in both local and wide-area networks [9]. The framework proposed in this paper differs from the MPICH-G2 approach because we don’t want to hide the topology, but empathize with it, and exploit this knowledge in order to choose the best communication technology in each program segment.

Stream processing has been widely studied to support high performance computing. Babcock *et al.* [2] and Gaber *et al.* [10] propose overviews on models and data mining using streams as well an overview of current projects, algorithms and techniques related to streaming models.

One of the most interesting application areas for streams is related to the use of dedicated processors, like GPUs [11]–[13]. While the major efforts focus on stream processor architectures [14], several studies exist on the use of such models in parallel systems. Wagner and Rostoker [15] propose an interesting stream-processing framework based on the MPI library: using the standard MPI structures they build a stream environment. To prove their approach, they apply it to a financial data analysis workflow. In this paper, instead, we use a whole MPI program as building block in a stream application, using it, not as communication framework, but only to speed-up heavy computational units, and exploit locality in clusters.

Financial data analysis is a hot topic in stream processing, although most of studies have focused on the use of single machines. For example Agarwal et al. [16] focus on financial analysis on multicore computers. There is also interest in how to adapt legacy code, written for streaming hardware architectures, to general purpose processors. Gummaraju et *al.*, in their inspiring paper [17], show how to map salient features of streaming models (e.g., kernels ...) to general purpose CPU components. The approach we propose in this paper, instead, exploits general-purpose parallel heterogeneous architectures.

Several studies focus on the languages and methodologies to describe streaming programming. Usually they propose the design of new languages [18], [19], or the extension of an existing one like C [12] or Java [20], [21]. Carpenter *et al.*, for example, in [6] propose an extension to the C language as a set of OpenMP-like directives. In our design, we have chosen a functional approach to be consistent with the MPI approach.

A similar approach is suggested, for example in [17].

VIII. Conclusions

The use of heterogeneous networks or "cluster of computational clusters" makes it difficult to reach optimal performance using only MPI programs. The impact of a slow link can strike the overall behavior, mainly when using collective operations. In this paper, we propose a way to exploit locality in complex heterogeneous computational systems using a hybrid approach of Stream and MPI programming models. We propose to model a parallel application as a chain of kernels following the "stream" approach, and to use MPI to develop the kernels that need more computational power.

We presented the prototype of a framework that supports the launch and the communication between sequential and MPI kernels. The evaluation on a financial application shows that the lighter overhead and minimal data structures in the stream portion of a hybrid MPI/Stream framework can improve processing time at certain order generation rates by as much as 58% compared to the corresponding MPI-only framework where the entire application is subject to higher middleware overhead.

In the future, we expect to support more communication infrastructure and to study how to describe the application in order to give the framework the hints needed to optimize the overall behavior of the application.

References

[1] MPI Forum, "Mpi: A message-passing interface standard," September 2009, http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," in *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM, 2002, pp. 1–16.

[3] K. Kandalla, H. Subramoni, G. Santhanaraman, M. Koop, and D. K. Panda, "Designing multi-leader-based allgather algorithms for multi-core clusters," in *Parallel and Distributed Processing Symposium, International*, vol. 0. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 1–8.

[4] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 requirements of real-time stream processing," *SIGMOD Rec.*, vol. 34, no. 4, pp. 42–47, 2005.

[5] E. P. Mancini, S. Marcarelli, I. Vasilev, and U. Villano, "A grid-aware mip solver: implementation and case studies," *Future Generation Computer Systems*, vol. 24, no. 2, pp. 133–141, February 2007.

[6] P. Carpenter, D. Rodenas, X. Martorell, A. Ramirez, and E. Ayguade, "A streaming machine description and programming model." in *Proc. of SAMOS'07 Conf.*, ser. Lecture Notes in Computer Science, S. Vassiliadis and et al., Eds., vol. 4599. Springer, 2007, pp. 107–116.

[7] ZeroMQ Stock Exchange Example, http://www.zeromq.org/code:examples-exchange.

[8] MPICH2 Message Passing Interface Standard, http://www.mcs.anl.gov/research/projects/mpich2/.

[9] N. T. Karonis, B. Toonen, and I. Foster, "MPICH-G2: A grid-enabled implementation of the message passing interface," *Journal of Parallel and Distributed Computing Special Issue on Computational Grids*, vol. 63, pp. 551–563, May 2003.

[10] M. M. Gaber, A. Zaslavsky, and S. Krishnaswamy, "Mining data streams: a review," *SIGMOD Rec.*, vol. 34, no. 2, pp. 18–26, 2005.

[11] J. H. Ahn, W. Dally, B. Khailany, U. Kapasi, and A. Das, "Evaluating the imagine stream architecture," in *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, June 2004, pp. 14–25.

[12] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*. New York, NY, USA: ACM, 2004, pp. 777–786.

[13] S. Yamagiwa and L. Sousa, "Design and implementation of a stream-based distributed computing platform using graphics processing units," in *CF '07: Proceedings of the 4th international conference on Computing frontiers*. New York, NY, USA: ACM, 2007, pp. 197–204.

[14] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens, "Programmable stream processors," *Computer*, vol. 36, no. 8, pp. 54–62, 2003.

[15] A. Wagner and C. Rostoker, "A lightweight stream-processing library using MPI," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 1–8, 2009.

[16] V. Agarwal, D. A. Bader, L. Dan, L.-K. Liu, D. Pasetto, M. Perrone, and F. Petrini, "Faster fast: multicore acceleration of streaming financial data." *Computer Science - R&D*, vol. 23, no. 3-4, pp. 249–257, 2009.

[17] J. Gummaraju and M. Rosenblum, "Stream programming on general-purpose processors," in *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 343–354.

[18] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language LUSTRE," in *Proc. IEEE*, vol. 79, no. 9, 1991, pp. 1305–1320.

[19] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: a system for programming graphics hardware in a c-like language," in *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*. New York, NY, USA: ACM, 2003, pp. 896–907.

[20] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek, "Streamflex: high-throughput stream programming in java," *SIGPLAN Not.*, vol. 42, no. 10, pp. 211–228, 2007.

[21] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *CC '02: Proceedings of the 11th International Conference on Compiler Construction*. London, UK: Springer-Verlag, 2002, pp. 179–196.

Appendix

For reference we show algorithm details for only the *Ask_Order_Routine*. The *Bid_Order_Routine* is identical except that it focuses on bid orders that are equal to, or greater than, the min_ask price.

Ask_Order_Routine(order)

Save current min_ask and max_bid for future reference.
loop
  if $order.price > max_bid$ **then**
    {*There are no matching buyers for this seller. Place the order on the tail of its price-appropriate double-ended queue in the OrderBook.*}
    $OrderBook[order.price]$.push_back($order$)
    {*Change min_ask value if needed.*}
    $min_ask \leftarrow$min(min_ask, $order.price$)
    Break from loop
  end if

  {*At this point order.price is less than or equal to max_bid. Get previously stored bid orders at max_bid price.*}
  $bid_deque \leftarrow OrderBook[max_bid]$
  $cumulative_volume \leftarrow 0$

  while bid_deque not empty **do**
    {*Execute a trade by matching current ask order with most recent bid order at this price level, and then send trade confirmation to bidder.*}
    $prev_bid \leftarrow bid_deque$.front()

$trade_volume \leftarrow$ min($order.volume$, $prev_bid.volume$)
send_trade_conf($prev_bid.id$, max_bid, $trade_volume$)

{*Adjust volume of prev_bid, ask order, and cumulative_volume to reflect this trade*}
$prev_bid.volume \leftarrow prev_bid.volume - trade_volume$
$order.volume \leftarrow order.volume - trade_volume$
$cumulative_volume \leftarrow cumulative_volume + trade_volume$

if $prev_bid.volume = 0$ **then**
{*Entire* $prev_bid$ *has been filled. Discard the bid.*}
bid_deque.pop()
end if

if $order.volume = 0$ **then**
{*Entire ask order has been filled.*}
Break from loop
end if
end while

if $cumulative_volume > 0$ **then**
{*Send a trade confirmation to the asker.*}
send_trade_conf($order.id$, max_bid, $cumulative_volume$)
end if

if $order.volume = 0$ **then**
{*Entire ask order has been filled.*}
Break from loop
end if

{*Any further iteration of loop will try to make trades at prices less than current max_bid.*}
$max_bid \leftarrow max_bid - 1$
end loop

repeat
Decrement max_bid by 1
until $OrderBook[max_bid]$.front() is not an empty queue or $max_bid = 0$

if max_bid or min_ask are different from the values saved at the beginning of the routine **then**
{*The market price has changed.*}
send_quote(max_bid, min_ask)
end if

if No trade confirmations were sent in the steps above **then**
{*Confirm the asker's order in the event that it cannot be immediately filled with a bid at current market conditions.*}
send_order_conf(order.id)
end if