# MB++: An Integrated Architecture for Pervasive Computing and High-Performance Computing

David J. Lillethun
davel@cc.gatech.edu

David Hilley
davidhi@cc.gatech.edu

Seth Horrigan
katana@cc.gatech.edu

Umakishore Ramachandran
rama@cc.gatech.edu

## Abstract

*MB++ is a system that caters to the dynamic needs of applications in a distributed, pervasive computing environment that has a wide variety of devices that act as producers and consumers of stream data. The architecture encompasses several elements: The type server allows clients to dynamically inject transformation code that operates on data streams. The transformation engine executes dataflow graphs of transformations on high-performance computing resources. The stream server manages all data streams in the system and dispatches new dataflow graphs to the transformation engine. We have implemented the architecture and show performance results that demonstrate that our implementation scales well with increasing workload, commensurate with the available HPC resources. Further, we show that our implementation can exploit opportunities for parallelism in dataflow graphs, as well as efficiently sharing common subgraphs between dataflow graphs.*

## 1 Introduction

Novel sensors, handhelds, wearables, mobile phones, and embedded devices enable the creation of imaginative pervasive computing applications to assist users in everyday environments. However, many such devices are constrained in processing capability, memory, and power consumption. In contrast, high-performance systems, such as clusters and grid resources, have much greater capabilities at the expense of size and mobility. While significant advances have been made in pervasive computing middleware, most solutions tend to be handcrafted for specific applications or specific environments. Many systems support pervasive applications with moderate computational requirements, but few are targeted for applications that are both pervasive and require full utilization of high-performance computing resources. A comprehensive solution requires not only facilities for managing data transport, but also support for managing and instantiating computation automatically.

By combining the plethora of new gadgetry with high-performance computing (HPC) resources, there is an opportunity to expand the scope of pervasive computing applications to domains such as emergency response and transportation that require computationally intensive processing for which the edge devices may not be sufficiently equipped. The following application scenario is intended to represent the more general class of applications that MB++ is designed to support: A metropolitan-area emergency response infrastructure may have data sources including traffic cameras, handheld or in-dash computers from local police, fire and burglar alarms in local buildings, as well as a variety of other roaming data sources. In addition to simply capturing such data and making it available to humans for monitoring, the application would also use HPC resources to more extensively analyze the incoming data in real-time to perform anomaly detection and potentially predict future problems by monitoring events in different locations. The addition of stronger data analysis requires harnessing a cluster or federated groups of HPC resources, and a mechanism for automatically managing computation.

Pervasive computing applications such as this would benefit from an infrastructure providing services that integrate solutions to these problems. In particular, this infrastructure should be designed to support applications that are both pervasive and require extensive processing, necessitating the efficient use of HPC resources (e.g. compute clusters). Such an infrastructure would leverage high-performance computing resources in order to transform data while it is being transported. *Transformations* are arbitrary computations on data streams, not limited to simple format conversions – common examples include data fusion, feature extraction, and classification. The scheduling and execution of transformations should be handled by the runtime and applications could be constructed in a straightforward manner through composition. With such an infrastructure, consumers of data could obtain the streams needed, when they are needed, and in the form desired, leveraging the ambient computing infrastructure composed of sensors and HPC resources.

Prior work in this area tends to be either focused on the pervasive computing side, or on the side of service composition that uses HPC resources (see Section 6 for more details). In our work, we explore the requirements of pervasive computing applications with a non-trivial need for HPC resources and present a general architecture for addressing such applications' needs.

MB++ is an infrastructure that allows pervasive computing devices to join and leave the system dynamically, and executes stream transformations on HPC resources. Some of the goals of the system are to support arbitrary transformations of data streams (including fusion), to allow clients to dynamically add transformation functions, and to schedule transformations in a manner balancing the workload of many consumers. Previous work addresses interoperability of devices with different communication requirements [10]. This architecture also builds on previous work addressing the problems of efficient data transport to consumers requiring diverse data types [14].
Specifically, we make the following contributions:

- An architecture that (1) allows clients to dynamically inject data types and transformation functions into the infrastructure, (2) enables the dynamic composition of transformations to create complex dataflow graphs, (3) executes transformations in dataflow graphs that require multiple inputs from different source streams, such as data fusion and feature extractors, (4) ensures safe execution of transformations by using sandboxing, and (5) manages available computational resources by scheduling the execution of dataflow graphs on HPC resources.
- A prototype implementation of the MB++ architecture and experimental evaluation of its capabilities.

Section 2 discusses the system requirements in detail. The architecture is presented in Section 3 and the MB++ implementation in Section 4. Section 5 gives an experimental analysis of the type and transformation systems, followed by related work in Section 6 and conclusions in Section 7.

## 2 Requirements

The nature of pervasive applications imposes a highly dynamic environment, both in terms of application needs and resource availability, which leads to an interesting set of requirements to be met by the infrastructure for managing stream data and transformations thereof. Recall the example application scenario presented in Section 1. One application within that environment may allow authorities to track a criminal during a burglary. Suppose the criminal trips an alarm while entering a store, which notifies the police. The police may have software that acts as a client to the pervasive system, and upon receiving this notification submits a request to the system to send feeds from all the cameras that can see the criminal to the authorities. This may require certain functions that are not already present in the system, such as a computer vision algorithm that detects people or an algorithm that uses a collection of motion detectors to find the precise location of activity in the building. First, the police would inject this code into the system. Then they would compose the injected components into an application that produces the desired result (i.e. tracking the criminal) and submit that request to the system. Since this application may have computationally expensive components (e.g. computer vision algorithms), the system assigns high-performance computing resources to run the algorithms. Finally, the police can read the output information from lightweight handheld or automotive devices while they are dispatched to the scene.

The MB++ infrastructure is meant to facilitate information exchange among the clients. At a minimum, it has to provide conduits for data communication between producers and consumers of information. Pervasive computing applications typically involve heterogeneous gadgetry that may need different data formats for the same source of data. For example, a video stream may be needed in Motion JPEG by the automotive display and in Java Media Format by a handheld device. Additionally, pervasive applications often require more than mere format conversions: Many compelling applications require continuous and highly-intensive feature extraction on multiple data streams. For applications that run on lightweight and mobile devices, the computation cost of these transformations may be prohibitive. Therefore, an infrastructure for pervasive applications must allow transformations to be run on high-performance resources not only for purposes of data format conversion, but also for feature extraction, data fusion, or any other arbitrary transformation algorithm.

As applications join the environment, the infrastructure must provide the data types and transformation methods required by these applications. Rather than restricting an application to a set of predefined transformations, which may not meet the application's needs, the infrastructure must allow clients to dynamically add and remove transformations. However, since this facility allows clients to inject code into the infrastructure, it is important that the code also be platform independent as well as safe.

In our prior work [14], we presented a minimal set of services, namely statically defined data format transformations by the producer of a stream. In this paper we present an architecture for dynamically injecting arbitrary transformation code, a general stream sharing framework, and an infrastructure for safely executing transformations on high-performance computing resources.

## 3 Architecture

Figure 1 shows the architectural elements of the system, which include: a *type server*, *stream server*, and *transformation engine*. The clients (producers and consumers of information) are on the edge of the network and constitute the pervasive computing environment. The rest of the architectural elements are expected to be hosted on HPC resources.

Client requests for adding and/or deleting transformations are routed to the type server (labeled 1 in Figure 1). When a client requests to execute a set of transformations on a stream, the stream server dynamically instantiates the

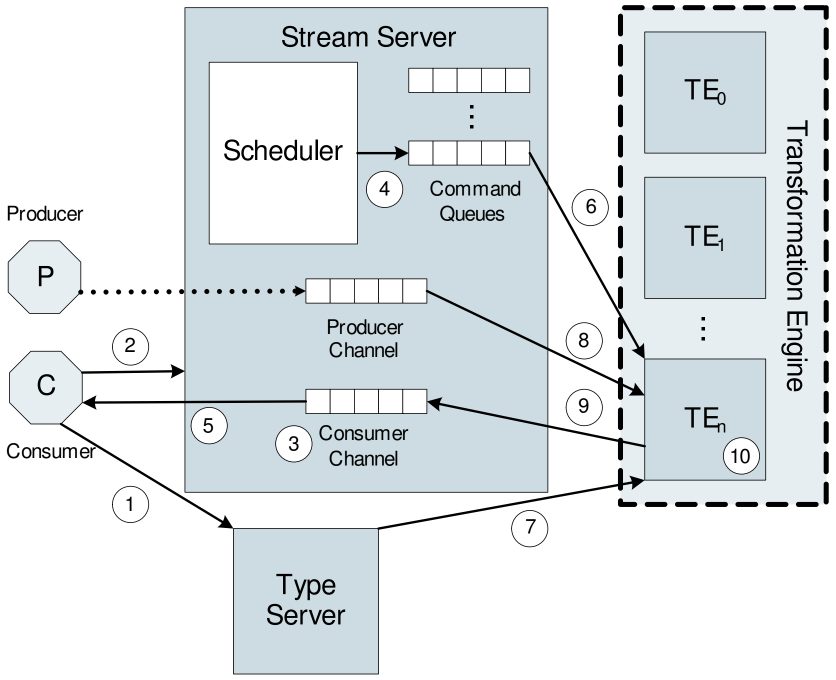

**Figure 1. Architecture and Control Flow**

necessary plumbing, in terms of the stream sources (labeled 2-9). The transformation engine executes the transformations on the available resources (labeled 10).

## 3.1 Type Server

Every stream has a *data type* that is indicative of both the structure and semantics of the data stream. A data type is expressed as a set of extensible user-defined attributes that describe the necessary aspects of the type. A *transformer* is the physical embodiment of a transformation. For example, face detection on a video stream is a transformation but a function that implements the face detection algorithm is a transformer. A transformer is expressed as a tuple consisting of the expected data types of the inputs, the code implementing the transformation, and the data type of the output, as shown in Figure 2. Data types provide information about which streams a transformer can operate on, and therefore about how transformers may be joined to create larger dataflow graphs.

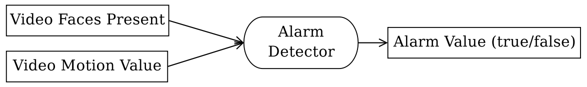

**Figure 2. Example Transformer**

The *type server* stores information about data types and transformations, and allows clients to dynamically add and remove both data type specifications and transformers. More than one transformation between the same set of input and output types may exist, provided there are distinct transformers (i.e. distinct code) qualifying each such relationship.

## 3.2 Transformation Engine

The *transformation engine* is the logical entity responsible for managing the available resources. The transformation engine comprises a number of *transformation environments* (TE) running on different HPC resources (such as nodes in a cluster). Each TE is an independent, multi-threaded process that runs on a single compute node and executes dataflow graphs assigned to it by the scheduler (described in the next section). The number of TE can be scaled up or down depending on the available resources and the need as demonstrated by the number of transformations to be performed. A dataflow graph running on a TE is a long-running entity that continues to process streams until the scheduler signals the TE to stop.

One of the important architectural roles of a TE is the safe execution of transformations. A TE serves to "sandbox" the transformers by separating them from the rest of the system. Depending on the implementation vehicle, the TE sandboxing can also conceal local resources, such as the file system and system calls that are not necessary for transformer execution.

The TE also provide platform independence to transformers, since transformer developers may not be able to predict the platform on which the transformer may be run.

### 3.3 Stream Server

The *stream server* is responsible for instantiating all streams as well as sending dataflow graphs to the transformation engine for execution. It includes a *scheduler* that determines on which resources each new transformation request should be run, and load balances dataflow graphs already running in the transformation engine.

A *dataflow graph* identifies the initial input streams (supplied by producers), plus any transformations to be applied to those streams, as shown in Figure 3. A consumer submits a dataflow graph to request transformed stream data from the stream server. Information in the type server allows verification of the data types in the dataflow graph.

Establishing the plumbing for a new dataflow graph involves the following steps as shown in Figure 1: Any client may submit data types and transformers to the type server (1); here the consumer is shown doing so prior to submitting a new dataflow graph. The client submits a dataflow graph by making the appropriate call to the stream server's API (2). If the dataflow graph requires transformation, the stream server establishes a new stream for the graph results, called a consumer stream (3), and schedules the transformations to be executed on a particular transformation environment by submitting the dataflow graph to the queue of commands issued to that TE (4). Then the stream server responds to the consumer with the connection information it will need to begin reading from the new consumer stream (5). Meanwhile, the transformation environment reads the transformation request from its command stream (6) and downloads the required transformer code from the type server (7). Then the TE connects to the producer streams that it will use for input (8) and the consumer stream where it will place the results of the transformation (9). Finally, the TE begins executing the transformations on the producer stream data and writing the output to the consumer stream (10). Once a TE begins executing a transformation request, it will continue to transform those streams until it receives a command to stop from the stream server.

Some dataflow graphs have a structure that allows certain portions to be executed in parallel, for example, the Face Detector and Motion Detector in Figure 3. To take advantage of this, the stream server can submit the parallelizable portions to the transformation engine such that they will be executed in parallel. It is also possible that two or more different dataflow graphs may have a subset of the graphs in common. In such a case, the common subgraph may be executed only once and the result shared between all dataflow graphs that have this subgraph in common.

## 4 Implementation

In this section, we describe the status of our current MB++ implementation. All the components of the prototype are implemented; however, we note where the implementation of a component lags behind the architectural specification laid out in the previous section.

### 4.1 Type Server

The type server is implemented in C++ and accessed remotely by a lightweight client library that communicates with the type server in an RPC-like fashion. The library exposes methods to add and remove data types and transformers, and to query and retrieve transformer code.

The type server back-end is implemented using a relational database. Storing data in memory fails to provide persistence, in case of a crash, shutdown, or migrating between machines. Writing data to disk, on the other hand, does not provide the features of a database, such as transactional semantics and a rich query language (i.e. SQL).

### 4.2 Stream Server

The *stream server* manages the necessary plumbing between the producers and consumers of the MB++ system for the execution of the dataflow graphs. It is implemented as a multi-threaded C++ runtime system, assuming an underlying infrastructure support for reliable timestamped data transport for the streams. The requirement for timestamped stream transport is met by a programming system called Stampede [13]. A Stampede program consists of a dynamic collection of threads communicating timestamped data items through *channels* and *queues*. Channels provide random access to items indexed by timestamps (including special wild card values for timestamps such as "get_latest" and "get_earliest"); queues support first-in first-out semantics for the items contained in them. The threads, channels, and queues can be launched anywhere in the distributed system, and the runtime system takes care of automatically garbage collecting the space associated with obsolete items from the channels and queues. D-Stampede [1] is the distributed implementation in C of the Stampede programming model, and allows Java, C, and C++ components of a given application to share channels and queues.

The stream server (implemented as a layer on top of the D-Stampede runtime) maintains a *command queue* for each transformation environment, implemented using Stampede queues. When a consumer submits a dataflow graph to the stream server, it follows the steps outlined in Section 3.3 to pass the graph on to a particular transformation environment using its command queue.

The stream server also includes a *scheduler*. Upon submission of new dataflow graphs, it simply enqueues them in the command queues of the TE in a round robin fashion. However, the scheduling algorithm can be changed easily without any change to the overall MB++ architecture.

An alternative to scheduling entire dataflow graphs would be to schedule each item as it is created by a producer. However, this approach introduces too much overhead into the critical path.

Currently, the scheduler does not perform any advanced analysis or verification of the dataflow graphs, nor does it do any load balancing across the TE. Parallelizable graphs may still be run in separate threads of execution within the

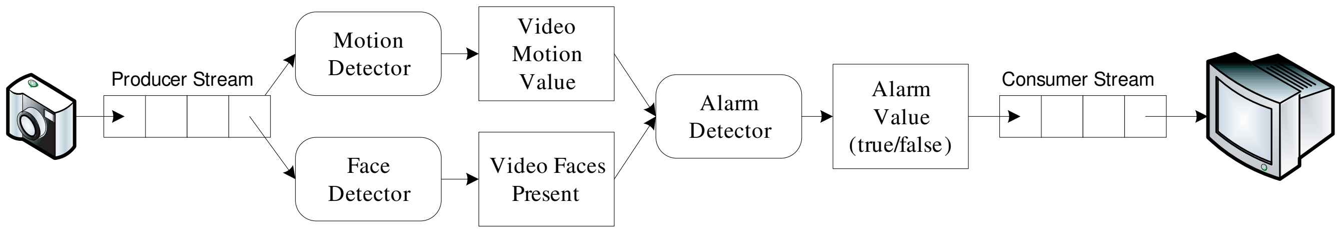

**Figure 3. Example Dataflow Graph**

transformation engine, but it is up to the client submitting the graph to explicitly specify how it is to be parallelized. The client may achieve this by submitting a dataflow graph in separate pieces, where each parallel branch is a distinct dataflow graph. Likewise, there is no automatic detection of dataflow subgraphs that may be shared between consumers. However, if two clients are aware of each other in advance, they may cooperate to create a sharing situation using the same mechanism.

### 4.3 Transformers

Transformers are Java classes that implement a common interface. Despite the superior performance of native code written in C or C++, Java is a better fit for the architectural needs for several reasons. First, the Java security model protects the MB++ system from hostile code which may perform unsafe activities such as local disk access, network communication, creating new processes, loading dynamic libraries, and directly calling a native method. Second, it provides platform independence so that a transformer may be implemented once and added to any MB++ instance, regardless of the underlying platform. Finally, the Just-In-Time byte-code compilation technique included in many modern JVMs reduces the performance disparity between Java and native code implementations of transformers.

### 4.4 Transformation Engine

The *transformation engine* is a conceptual aggregation of available computational resources for running transformations in parallel on HPC resources. Each participating node runs a container instance called a *transformation environment* (TE). A TE runs in a Java Virtual Machine (JVM) and executes transformers represented as Java byte-code.

A TE has a main thread of execution whose sole responsibility is handling commands retrieved from the TE's command queue. When a command requests that a new dataflow graph be instantiated, the necessary transformer code is retrieved from the type server and the transformers are instantiated. Then a new Java thread is created to execute the dataflow graph. While the scheduling of Java threads is JVM implementation dependent, some common configurations, such as the Sun JVM on Linux for SMP, allow true concurrency on multiprocessor systems. Therefore, a transformation environment may execute dataflow graphs concurrently on a multi-processor/multi-core architecture if the underlying system (JVM and OS) has SMP support.

## 5 Performance

To demonstrate the performance of MB++ we present three experiments. The first is a set of microbenchmarks for the most common type server requests. The next shows the scalability of the transformation engine as the number of dataflow graphs are varied. The final experiment demonstrates the performance improvement gained by parallelizing dataflow graphs and sharing common subgraphs.

The type server runs on a RedHat Enterprise Linux 4 (RHEL4) machine using the Linux 2.6.9 SMP kernel with two hyperthreaded 3.20 GHz Intel Xeon processors. The stream server and each of the TE execute on cluster nodes that are RHEL4 machines using the Linux 2.6.9 SMP kernel with two hyperthreaded 3.06 GHz Intel Xeon processors. The cluster's internal network is Gigabit Ethernet. All producers and consumers are run on a RHEL4 machine using the Linux 2.6.9 SMP kernel with two hyperthreaded 3.06 GHz Intel Xeon processors.

For the latter two experiments, two video producers are used that read video files and place the video frames into their streams. Both producers execute the same code, but read from different video files. In addition, they add the Face Detection and Motion Detection Transformers to the type server prior to streaming video. Producers of live video streams are also developed, but we chose to use prerecorded video in order to make the experiment more reproducible.

The Motion Detection Transformer determines the amount of motion in a video frame and outputs that value. To accomplish this, it keeps state (independently for each instance of the transformer) that contains the pixels of the previous frame, which it compares to the current frame.

The Face Detection Transformer determines the number and position of any faces in a video frame. To accomplish this, we use Intel's OpenCV library which is not available as Java byte-code. There is an executable daemon written in C, which is already present on the nodes running TE, that receives video frames via IPC and returns the face information. For security reasons, this would not be possible on a real deployment of the system, and Java libraries would need to be used to support transformers.

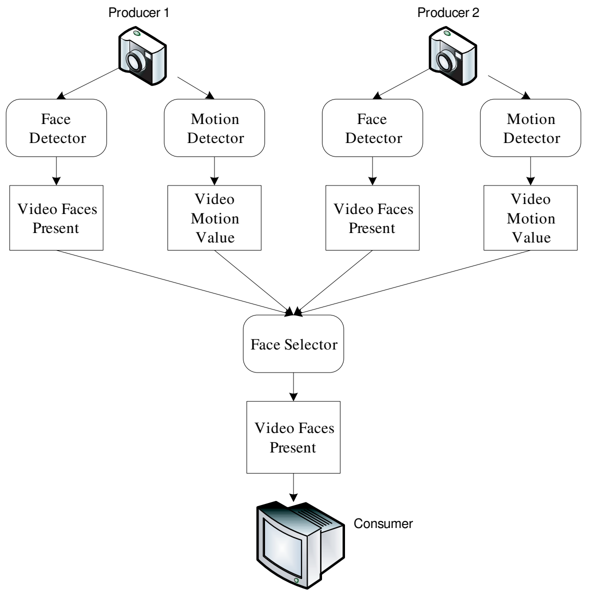

**Figure 4. Experiment Dataflow Graph**

| API Call | Time (ms) | Std. Dev. |
|---|---|---|
| *add_type* | 80.8 | 0.619 |
| *add_transform* | 82.9 | 4.107 |
| *query_transforms* | 79.9 | 0.031 |

**Table 1. Type Server Benchmarks**

Consumer clients submit the dataflow graph shown in Figure 4, but submit the graph differently in each experiment in order to execute the entire dataflow graph in serial, in parallel, or with shared subgraphs. The consumers also provide the Face Selection Transformer to the type server.

The Face Selection Transformer takes two pairs of face and motion streams as inputs. It selects the video with the most motion (largest motion stream value), and returns the face information from the corresponding face stream.

## 5.1 Type Server Latency

The latency microbenchmarks are recorded using a program that acts as a client to the type server and measures the end-to-end latency for the three most common requests. In order to eliminate the variable effect of network latency, the client was executed on the same system as the type server using the localhost loopback IP address.

Table 1 shows the latency associated with the requests. These calls are regularly used by producers when registering types and by consumers when adding new transformers to process streams. However, they are not in the critical path of stream processing in the transformation engine.

As the data demonstrates, clients can add tens of types and transformers per second, which is acceptable since they are only called at set-up time. In the tests, a very small code section of 12 bytes was used. The actual *add_transform* time will increase with the size of the code contained within the transformer, mainly due to network latency. Similarly, network latency may affect the time to service a *query_transforms* request depending on the number of transformers that are returned.

## 5.2 Dataflow Graph Scaling

To demonstrate scaling as the number of dataflow graphs increases, we measure the end-to-end latency of the execution of a dataflow graph. It is important to note that this includes not only the time needed to execute the graph in the transformation engine, but also the network communication. Each measurement is taken at the producer just before the data item is put in the Stampede channel, and on the consumer just after the item is retrieved. The difference in these measurements is the latency to transform a single item (e.g. a video frame). The data points presented below are the average of the latencies encountered while transforming 180 items. In cases with multiple consumers, the data point represents the average across all consumers.

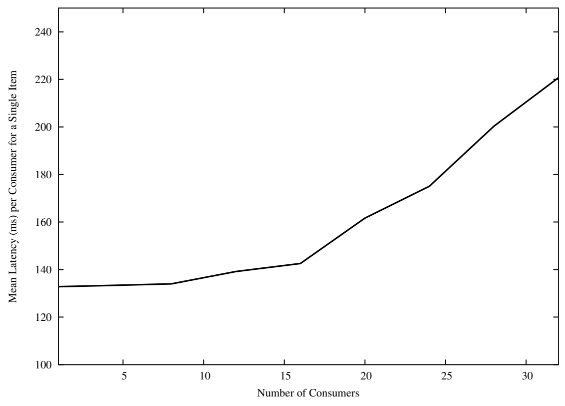

**Figure 5. Performance Scaling of Serial Dataflow Graphs**

This experiment uses two video producers, as described earlier, and the transformation engine is comprised of 8 TE, each running on a separate SMP cluster node. A variable number of consumer clients submit separate but identical dataflow graphs, as shown in Figure 4. The graphs from different consumers are executed in parallel with each other, but each individual graph executes its transformers serially.

The results are shown in Figure 5, where each data point represents the mean latency of a single item. Little change in latency is expected while the number of dataflow graphs is less than or equal to the number of processors in the transformation engine. This is in fact what the results show, as the latency change between 1 consumer and 16 consumers is only 10ms. However, latency begins to degrade as the computational resources become loaded, as demonstrated by a 55% increase (78ms) in latency from 16 to 32 consumers.

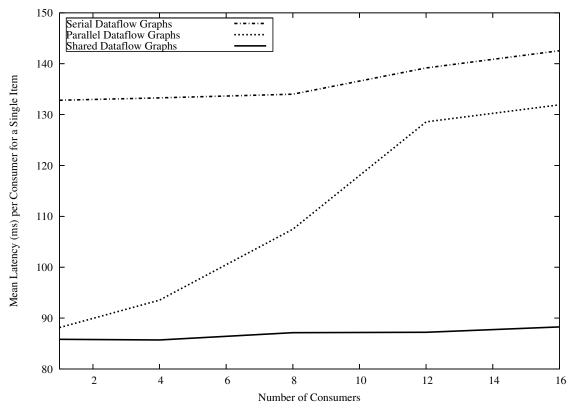

**Figure 6. Performance Scaling of Parallel and Shared Dataflow Graphs**

## 5.3 Dataflow Graph Parallelization

This experiment demonstrates the benefit of parallelizing dataflow graphs. The setup is the same as the previous experiment except that the consumer clients submit the dataflow graphs so that the transformation engine executes portions of each graph in parallel. Each consumer submits the entire graph, so that the total number of transformations being executed is still five per consumer, as shown in Figure 4. As a side effect, sequential data items are also partially pipelined, since the Motion Detection and Face Detection Transformers can compute item $i+1$ in parallel with the Face Selection Transformer computing item $i$.

The performance improvement from parallelizing the dataflow graphs should be substantial as long as the computational resources are not over-utilized. However, as the load becomes heavy, the benefit should decrease since the system resources' limited capacity to do the work becomes the dominant factor. Figure 6 verifies this hypothesis by comparing the performance of the parallelized dataflow graphs with the serial version of the same graphs from the previous experiment. With 4 consumers submitting dataflow graphs to the eight transformation environments, there is a 30% decrease in latency by parallelizing the graphs. However, when the system is loaded with 12 consumers trying to run 60 transformers in parallel, there is only a 7.6% decrease in latency.

## 5.4 Dataflow Subgraph Sharing

The final experiment shows how consumers with common dataflow subgraphs can collude to reduce latency by sharing these subgraphs. The consumer clients are similar to those used in the dataflow parallelization experiment, except that each of the first four subgraphs (the face and motion detection on the two video streams) are instantiated only once. Each consumer submits its own face selection dataflow graph, but each instance takes input from the same four shared subgraphs. Therefore, for $N$ clients the number of transformations being executed is reduced from $5N$ to only $N+4$.

The primary difference between the shared and parallel cases is that the amount of computation required increases more slowly as additional consumers are added. With 16 consumers, the serial and parallel cases are both taxing the available resources with 80 total transformations, while the subgraph sharing case only places 20 transformations on the sixteen processors. Therefore the latency should increase only after many more consumers join the system than in the parallel or serial experiments.

Figure 6 confirms this by comparing the shared dataflow graph latency to the parallel and serial experiments. The shared experiment shows only a 2.8% increase in latency as the number of consumers increases from 1 to 16. With 16 consumers, the shared graphs have a latency of only 88ms, while the parallel and serial experiments have latencies of 132ms and 143ms, respectively.

## 6 Related Work

Pieces of the architectural framework of MB++ can be found in other related work; however, a composite architecture that represents a true marriage of HPC and the pervasive computing environment as envisioned in MB++ does not exist to the best of our knowledge.

Much work in stream-handling middleware focuses on either multimedia streams or sensors streams. IrisNet [11] is a middleware designed to support database-oriented, sensor-rich Internet applications, a subset of applications relevant to MB++'s domain. The Distributed Media Journaling project [6] provides a middleware to support the live indexing of media streams, defining applications in terms of three types of component transformations (filters, feature extractors, or classifiers) operating on media sources or other transformation output streams.

There is a large body of work in more declarative and domain-specific stream processing systems using dataflow graphs and transformation operations. TelegraphCQ [2], a continuous query database system, uses a combination of SQL and a lower-level query language to express sliding window based continuous queries over streaming data sources. Other streaming database systems present stream manipulation via SQL but do not utilize continuous queries, such as Gigascope [5]. Many domain-specific and higher-level distributed programming languages provide elements of stream processing, such as Spindle [4] and Sawzall [12] (though Sawzall operates on stored data sets rather than streams) . The actual data processing mechanism is just a piece of the MB++ approach and we see such declarative processing work as complementary to our goals.

Solar [3] is a middleware for event-driven, context-aware applications composed of *event streams* and fusion operators. MB++ supports streams of arbitrary data, thus allowing not only event streams but also other types of data streams, such as media streams. Solar also processes streams in distributed *planets* rather than using HPC resources like MB++. The Ninja Architecture [7] is a large project encompassing a range of goals related to service-oriented applications. Services are provided by high-availability computational resources, like a cluster, called *bases*. However, like Solar, Ninja's primary mechanism for processing streaming data is an overlay network of *active proxies*.

There is also extensive work related to delivering streaming media content where the streams are transformed by service overlay networks or dynamically configured composable services in order to meet varying QoS demands of consumers [8, 9, 15]. Many of these works also cite the aforementioned Ninja Architecture as a foundational effort in *service composition* as a mechanism for creating distributed applications for this domain. All of these frameworks and systems achieve similar goals as MB++ but the applications these systems tend to support have different characteristics. Applications are created by dynamically composing existing available services with interposed adapters to resolve data format and protocol mismatches. This approach tends to favor more loosely-coupled applications with greater geographic distribution, while MB++ favors more tightly-coupled and potentially higher performance applications.

Our own prior work focused on data stream registration and discovery, and on transformation of data to different formats and fidelities [14]. However, both the architecture and the implementation of our earlier system were limited in scope: (a) the architecture did not support dynamic injection of arbitrary transformations or fusion of multiple streams, and (b) the implementation did not fully exploit HPC resources for executing dataflow graphs.

## 7 Conclusions

MB++ is an infrastructure that establishes a union between pervasive computing devices and high-performance computing resources in dynamic pervasive computing environments. At the heart of the architecture is a unique capability for the dynamic injection, composition, and execution of stream transformations. The system has been implemented and our experimental results show that performance scales with the number of dataflow graphs being executed and the amount of computational resources available.

There are a number of avenues for future research. An online analysis of dataflow graphs submitted to the stream server would allow automatic detection of parallelizable portions of the graph, as well as subgraphs that can be shared with other graphs being executed. Adding priority to dataflow graphs would allow performance commensurate with their perceived importance. Federating the architecture would allow multiple MB++ instances to work together and share streams. Finally, deploying MB++ for a specific real-world application would provide valuable insight into its limits and capabilities.

### 7.1 Acknowledgments

The work has been funded in part by an NSF ITR grant CCR-01-21638, NSF NMI grant CCR-03-30639, NSF CPA grant CCR-05-41079, and the Georgia Tech Broadband Institute. The equipment used in the experimental studies is funded in part by an NSF Research Infrastructure award EIA-99-72872, and Intel Corp. We thank the members of the Embedded Pervasive Lab at Georgia Tech (http://wiki.cc.gatech.edu/epl/) for their helpful feedback on our work.

## References

[1] S. Adhikari, A. Paul, and U. Ramachandran. D-Stampede: Distributed programming system for ubiquitous computing. In *Proceedings of ICDCS'02*, July 2002.

[2] S. Chandrasekaran et al. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of CIDR '03*, January 2003.

[3] G. Chen and D. Kotz. Solar: An open platform for context-aware mobile applications. In *Proceedings of the First International Conference on Pervasive Computing*, pages 41–47, June 2002. Short Paper.

[4] C. Consel et al. Spidle: a dsl approach to specifying streaming applications. In *Proceedings of GPCE '03*, pages 1–17, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[5] C. Cranor et al. Gigascope: A Stream Database for Network Applications. In *Proceedings of SIGMOD '03*, pages 647–651, New York, NY, USA, 2003. ACM Press.

[6] V. Eide, F. Eliassen, and O. Lysne. Supporting distributed processing of time-based media streams. In *Proceedings of DOA'01*, pages 281–288, September 2001.

[7] S. D. Gribble et al. The Ninja architecture for robust Internet-scale systems and services. *Computer Networks*, 35(4):473–497, 2001.

[8] J. Jin and K. Nahrstedt. QoS Service Routing for Supporting Multimedia Applications. Technical Report IRP-TR-03-04, Department of Computer Science, University of Illinois at Urbana-Champaign, November 2002.

[9] J. Liang and K. Nahrstedt. Service composition for advanced multimedia applications. In *Proceedings of MMCN'05*, pages 228–240, 2005.

[10] J. Nakazawa et al. A bridging framework for universal interoperability in pervasive systems. In *Proceedings of ICDCS'06*, July 2006. to appear.

[11] S. Nath et al. IrisNet: An architecture for enabling sensor-enriched internet service. Technical Report IRP-TR-03-04, Intel Research Pittsburgh, June 2003.

[12] R. Pike et al. Interpreting the Data: Parallel Analysis with Sawzall. *Scientific Programming*, 13(4):277 – 298, 2005.

[13] U. Ramachandran et al. Stampede: A cluster programming middleware for interactive stream-oriented applications. *IEEE TPDS*, 14(11):1140–1154, November 2003.

[14] U. Ramachandran et al. Mediabroker: A pervaisve computing infrastructure for adaptive transformation and sharing of stream data. *Pervasive and Mobile Computing*, 1(2):257–276, July 2005.

[15] D. Xu and X. Jiang. Towards an integrated multimedia service hosting overlay. In *Proceedings of ACM Multimedia '04*, pages 96–103, New York, NY, USA, 2004. ACM Press.