A Pipelining Implementation for Parsing X-ray Diffraction Source Data and Removing the Background Noise

To cite this article: Michael A Bauer et al 2010 J. Phys.: Conf. Ser. 256 012017

View the article online for updates and enhancements.
A Pipelining Implementation for Parsing X-ray Diffraction Source Data and Removing the Background Noise

Michael A. Bauer*, Alain Biem†, Stewart McIntyre* and Yuzhen Xie*

*University of Western Ontario, London ON, Canada
†IBM Research, Yorktown Heights, United States
E-mail: bauer@uwo.ca; biem@us.ibm.com; smcintyr@uwo.ca; yxie@csd.uwo.ca

Abstract. Synchrotrons can be used to generate X-rays in order to probe materials at the atomic level. One approach is to use X-ray diffraction (XRD) to do this. The data from an XRD experiment consists of a sequence of digital image files which for a single scan could consist of hundreds or even thousands of digital images. Existing analysis software processes these images individually sequentially and is usually used after the experiment is completed. The results from an XRD detector can be thought of as a sequence of images, generated during the scan by the X-ray beam. If these images could be analyzed in near real-time, the results could be sent to the researcher running the experiment and used to improve the overall experimental process and results. In this paper, we report on a stream processing application to remove background from XRD images using a pipelining implementation. We describe our implementation techniques of using IBM Infosphere Streams for parsing XRD source data and removing the background. We present experimental results showing the super-linear speedup attained over a purely sequential version of the algorithm on a quad-core machine. These results demonstrate the potential of making good use of multi-cores for high-performance stream processing of XRD images.

1. Introduction

Synchrotrons are large electron storage rings that produce X-ray beams of unprecedented brightness. Synchrotron facilities, such as the Canadian Light Source (CLS) in Saskatchewan and the Advanced Light Source (ALS) in the US, allows scientists to use synchrotron beams to study various materials. In order to understand and predict properties of materials, detailed information about the structure of the material at the atomic-scale level is needed. Synchrotrons can be used to generate X-rays in order to probe materials at the atomic level. One approach is to use X-ray diffraction (XRD) to do this. The Laue X-ray diffraction (XRD) technique has been used to determine the atomic-scale structures of materials based on the Bragg's Law [17]. When a material that consists of a periodic atomic structure, such as a crystal, is irradiated with X-rays, it produces a diffraction pattern showing numerous sharp spots, called Bragg diffraction peaks. By measuring and analyzing the positions and intensities of these peaks, one can determine the arrangement of atoms in the crystalline material. Not only can Laue diffraction provide geometrical information about the crystalline structure, it can also be extended to measure the intensity of X-ray diffraction to determine the strain tensor in sample materials [4]. The strain tensor information will then help scientists/engineers to determine possible flaws in materials.
The data from an XRD experiment consists of a sequence of digital image files. The XRD data analysis software actually deals with a large amount of data, typically, digital images in the hundreds or even thousands. Existing analysis software processes these images individually sequentially. This is a very time consuming process, normally requiring days to complete the processing of an entire set of data from one XRD experiment. A number of existing software packages for XRD data analysis have been developed. These include the 3D X-ray Micro-diffraction Analysis Software Package [16] at the Advanced Photon Source (APS), which was developed by scientists at the Oak Ridge National Laboratory, and the X-ray Micro-diffraction Analysis Software (XMAS) [15] at the Advanced Light Source (ALS) in Lawrence Berkeley National Laboratory (LBNL). The common feature of these two packages is that they both are Windows-based software, implemented in the Interactive Data Language (IDL) [10]. Both can process a large amount of XRD data sequentially.

Synchrotron time is valuable and it is often difficult to get a scheduled beam time. Data analysis using existing software means that researchers are typically doing the analysis after their time on the synchrotron. More timely analysis could help researchers make decisions on subsequent experiments and gain significant insight into the materials that they are studying. Performance improvement in the analysis software could make the analysis more useful.

Synchrotron experiments can produce several megabytes of image data per second. One can think of the output from such an experiment as a “stream of images”. As part of a project funded by Canada’s Advanced Research and Innovation Network (CANARIE), we are involved in a project to study the feasibility of processing such “streams” of images. The goal of the Active Network Interchange for Scientific Experimentation (ANISE) project is to develop a high speed network and processing systems to achieve near real-time data analysis for synchrotron experiments. Being able to process this data in near real-time would enable users to make sound decisions about issues on time-sensitive materials under study.

Our proposed approach is essentially one of stream processing. The objective is to be able to stream images, as they are generated, to a processing center across CANARIE’s network and return results to a user. In this paper we report on our initial efforts in developing stream processing software capable of analyzing such images. Our development efforts make use of IBM InfoSphere Streams, a software environment for developing streaming applications that can receive and process data in multiple streams and in parallel. Our initial efforts have been on the development of software to perform background filtering on XRD images, one of the central computational steps in the analysis of XRD images. We describe our techniques for developing stream processing components and our experiences with InfoSphere Streams. We report the performance of our program for processing real-time problems of different sizes.

The paper is organized as follows. In Section 2 we provide a brief introduction and an overview of our stream processing platform - InfoSphere Streams and its development environment SPADE. In Section 3 we introduce the sequential algorithm for the two dimensional (2D) background removal. In Section 4 we outline our pipeline implementation in some detail and include a number of diagrams illustrating how the background processing is decomposed into elements in order to be processed in a pipelined manner. Experimental results are presented in Section 5 and we provide some concluding comments in Section 6.

2. Stream Processing Platform

Stream processing is a data-centric programming model. It applies to applications where the data to the application can be viewed as a stream of inputs and where the application may entail a limited form of parallel processing. Given a set of data which can be structured as a stream of tuples, a series of operations (processing elements) are applied to each tuple or some set of tuples in the stream. Processing elements are usually pipelined, much like an assembly line. Pipelining does not reduce the time for individual instruction execution. Instead, it improves instruction
and data throughput. The perfect condition is that all the pipe stages, i.e. processing elements, are balanced. In such case, the speedup from pipelining equals the number of pipe stages.

Our implementation platform is IBM InfoSphere Streams [7] targeting clusters of multicore. Streams is an emerging software platform in support of parallel and high-performance stream processing. It consists of an application development framework called Stream Processing Application Declarative Engine (SPADE) [9] and a runtime system that can facilitate the running of SPADE applications on a distributed set of hosts.

In SPADE, a stream processing application is modeled as a dataflow graph. It is composed of data streams and operators. A data stream is a running sequence of tuples. A tuple is an individual piece of data in a stream. Typically, the data in a tuple represents the state of something at a specific point in time. For example, a row of pixel values in an XRD image, say produced by an XRD detector while scanning on an area of the material being tested, could be considered a tuple. An operator takes in a stream, manipulates the tuple data and generates the results in a form of an output stream. Multiple operators can work on different tuples of a stream or multiple streams. Figure 1 shows a dataflow graph for a SPADE application example [8]. The icons in the figure represent operators, and the lines represent streams.

![Figure 1: A dataflow graph for a simple SPADE application](image)

When a SPADE application is compiled, the operators and streams are organized into a set of individual execution units called Processing Elements (PEs). The application’s structure is described in a Job Description Language (JDL) file, including details about each PE, such as which binary file to load and execute, command-line parameters, scheduling restrictions, stream formats, and internal operator dataflow graph. To run an application, a user needs to submit a job to the Streams runtime system. The Streams runtime system provides a collection of services such as Resource Manager, Scheduler and Host Controller to manage the running of SPADE applications on a distributed set of hosts. Detailed report on Streams runtime system is referred to [8, 9].

2.1. Streams operators

As a programming interface for end-users to operate on data streams, Streams makes available a toolkit of built-in operators to manipulate data streams. A subset of operators currently supported by Streams include:

**Source**: A Source operator is used for creating a stream of data from an external source.

**Sink**: A Sink operator is used for converting a stream into a flow of tuples that can be used by components that are not part of Streams.

**Functor**: A Functor operator is used for performing tuple-level manipulations such as filtering, projection, mapping, attribute creation and transformation.

**Aggregate**: An Aggregate operator is used for grouping and summarization of incoming tuples. This operator supports a large number of grouping mechanism and summarization functions though sliding or tumbling windows.

**Join**: A Join operation is used for correlating two streams.
**Split**: A Split operator is used for splitting a stream into multiple output streams, based on a split condition.

**Barrier**: A Barrier operator is used as a synchronization point.

SPADE is equipped with a facility to extend the underlying system with user-defined operators (UDOPs), which are operators specialized to both wrap legacy libraries and provide customized processing. In Section 4 we will introduce the UDOPs that we have created in our application, as well as the built-in operators that are used to structure the data streams.

### 3. The Sequential Procedure

We describe a set of procedures for parsing X-ray diffraction (XRD) source data and removing the background noise. XRD is one of the important measurements by X-ray radiation with numerous applications [5]. The high-intensity synchrotron X-ray sources allow diffraction experiments to be performed on very small crystals or on large biological molecules. Background can be produced from a number of sources such as fluorescent radiation emitted by the specimen, diffraction of a continuous spectrum of wavelengths, etc. Estimation of background is a primary requirement for many types of further analysis.

There are different approaches for estimating and removing the background from X-ray diffraction patterns. For example, Kajfosz and Kwiatek [11] utilize a non-polynomial method for approximation of background in X-ray spectra. A polynomial filter [6] is used by Dinnebier and the smoothing procedure [2] was introduced by Brückner for 1D powder diffraction patterns. These methods are usually extended to treat 2D XRD images. For instance the combination of Brückner's 1D procedure and the 2D median filter utilized in XMAS [14, 15] for fitting the background of 2D X-ray microdiffraction patterns. In this work, we consider a practical algorithm which uses a parabolic filter to estimate the background of a 2D XRD image, given in Algorithm 1. It is an extension of the polynomial method for 1D powder diffraction patterns to handle 2D XRD images. Based on the knowledge provided by [16], it has been implemented as a sequential C procedure [12] in the Science Studio project [13].

**Algorithm 1: Remove 2D XRD background using a parabolic filter [12]**

<table>
<thead>
<tr>
<th><strong>Input</strong></th>
<th>file $F$ encoding a 2D XRD image $S$ of dimension sizes $r \times c$, a resizing factor $e$, a filter size $w$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Output</strong></td>
<td>image $S$ freed of background noise</td>
</tr>
</tbody>
</table>

1. $S \leftarrow$ Parse($F$);
2. $N \leftarrow$ Resize($S, r/e, c/e$);
3. $M \leftarrow$ GenerateFilter($w$);
4. $R \leftarrow$ ComputeRatio($P, M$); \hspace{1cm} // $O(r \ c \ (\frac{w}{e})^2)$
5. $B \leftarrow$ EstimateBackground($R, M$); \hspace{1cm} // $O(r \ c \ (\frac{w}{e})^2)$
6. $B \leftarrow$ Resize($B, r, c$);
7. $C \leftarrow$ Subtract($S, B$);
8. return $C$;

In Algorithm 1, the source of the 2D XRD image is from a file $F$. The function Parse reads the data from $F$ and converts it to a $r \times c$ matrix $S$ of energy intensity values. The data type of energy intensities is double. Usually the size of $S$ is large, for instance, $2084 \times 2084$. In order
to reduce the cost in removing the background, one can first scale down the dimension of $S$ by a factor of $e$. Let the smaller image be $N$ with dimension sizes $s \times t$. Thus we have $s = r/e$ and $t = c/e$. The functions in Lines 3 to 5 will work on $N$ and estimate its background.

In Line 3 of Algorithm 1, a filter or mask defined by a square matrix $M$ of order $c$, is generated by a parabolic function which models the distribution of the background noise in the neighborhood of a pixel. The mask $M$ is then applied to each pixel in $N$ for calculating the minimal ratios $R$ (a matrix of $s \times t$) by the function ComputeRatio. In this operation, for each pixel $(i, j)$ in $N$, with $0 \leq i < s$, $0 \leq j < t$, we compute the following minimum ratio

$$r_{i,j} := \min \left\{ \frac{n_{i+h,j+k}}{m_{f+h,g+k}} \mid \begin{aligned} -f &\leq h < w - f \\ -g &\leq k < w - g \\ -i &\leq h < r/e - i \\ -j &\leq k < c/e - j \end{aligned} \right\},$$

(1)

where $(f,g)$ are the coordinates of the centre of $M$.

After obtaining the minimum ratios, we apply the mask $M$ to each point in $R$ to estimate the background by EstimateBackground. To do so, for each point $(i, j)$ in $R$ $(0 \leq i < s$, $0 \leq j < t)$, we compute the following maximum product

$$b_{i,j} := \max \left\{ r_{i+h,j+k} \cdot m_{f+h,g+k} \mid \begin{aligned} -f &\leq h < w - f \\ -g &\leq k < w - g \\ -i &\leq h < r/e - i \\ -j &\leq k < c/e - j \end{aligned} \right\}.$$  (2)

The smallest value between $b_{i,j}$ and $n_{i,j}$ is the background of the pixel $(i,j)$ in $N$. Once the background values $B$ are computed, we increase its dimension sizes to the original ones. At the end, we subtract (pixel-wise) the background from the source image $S$.

It is trivial to prove that ComputeRatio and EstimateBackground are the most costly operations in Algorithm 1. For each, the number of operations in terms of $+, \times, /$ and comparison is bounded by $O(r c (\frac{w}{e})^2)$.

4. Pipelining Implementation

In this section we present our design and implementation of a fine-grained pipelining version of Algorithm 1 in SPADE. According to the stream processing model introduced in Section 2, we structure the data involved in the computations in Algorithm 1 into streams and organize the operations into operators which can work concurrently on elements of data (set of tuples) in the streams. Please refer to Algorithm 2 in Appendix for our SPADE implementation.

The list of sub-figures in Figure 2 illustrates the data streams and operators of Algorithm 2 with an example. In this example, the width of the image is $c = 10$. The resizing factor is $e = 2$ and the order of the parabolic filter is $w = 3$. The quantities $c, e, w$ are defined in Section 3. The steps in the following description correspond to the operations in Algorithm 2.

(1) First, a user-defined source operator Parse is used to parse the image data from the input binary file and turn it into tuples of intensity values of type double, as shown in Figure 2 Subfigure (1). A tuple represents a row of the input 2D image data. The sequence of those tuples forms a stream called Stream.s. In this example, as mentioned above, the number of pixels in each tuple is 10.

(2) A built-in operator Aggregate is used to make a tumbling window, called Window_A of size $e = 2$ from Stream.s. The Aggregate operator of SPADE aggregates a number of tuples from the incoming stream into one tuple (like a window) of the output stream. Figure 2 shows a state where Window_A consists of two input tuples $\{s_{00}, \ldots, s_{09}\}$ and $\{s_{10}, \ldots, s_{19}\}$.

(3) Each Window_A tuple is passed to Resize, a user-defined operator (UDOP), so as to scale down each dimension size of the image by a factor of $e = 2$. This generates Stream.n where each tuple has 5 data elements. For instance, at the state shown in Figure 2, the Resize
**Parse**: read data from an XRD source file block by block and convert them to intensity values. Each row of 10 values is organized as a tuple. Tuples are generated one by one, forming a stream, called Stream_s. One tuple (row) is generated at this state.

![Diagram of Algorithm 1](image_url)

**Window_A**: group every 2 tuples from Stream_s into 1 tuple.

![Diagram of Algorithm 1](image_url)

Figure 2: Stream processing model of Algorithm 1 (Subfigures (1) and (2)).
(3) Resize: take in a Window_A and scale down the 2 tuples of size 10 into a tuple of size 5, generating Stream_n. E.g. \( n_{04} = (s_{08} + s_{09} + s_{18} + s_{19}) / 4 \).

(4) Sliding_Window_B: inject Stream_n and create windows with each having 3 tuples and a sliding factor of 1. E.g. the 1st window contains Tuples 0, 1 and 2; The 2nd window contains Tuples 1, 2 and 3.

Figure 2: Stream processing model of Algorithm 1 (Subfigures (3) and (4)).
(5) **Compute Ratio**: for each pixel in a Stream\_n tuple, compute the minimum ratio among its neighbours w.r.t a mask M of size 3x3 with the center of M, m_{11}, over this pixel.

E.g. r_{04} = \min \{ n_{03}/m_{10}, n_{04}/m_{11}, n_{13}/m_{20}, n_{14}/m_{21} \}.

(6) **Sliding Window C**: take in Stream\_r and create windows with each having 3 tuples and a sliding factor of 1.
(7) **Compute Background**: for each pixel in Stream \( r \), define its background value as the maximum product of the ratio and the mask value among its neighbours with the center of mask, \( m_{ij} \), over this pixel. E.g. \( b_{04} = \max \{ r_{03}m_{10}, r_{04}m_{11}, r_{13}m_{20}, r_{14}m_{21} \} \). One tuple of 5 background values is then scaled up to 2 tuples of size 10, which are submitted to Stream \( b \).

(8) **Barrier**: synchronize and bundle Stream \( s \) and Stream \( b \) in order to match the original pixel values and their corresponding background values.

Figure 2: Stream processing model of Algorithm 1 (Subfigures (7) and (8)).
operator injects a Window_A tuple with data \{s_{10}, \ldots , s_{19}\}. It then computes
\(n_{00} = (s_{00} + s_{10} + s_{01} + s_{11})/4, \ldots , n_{04} = (s_{08} + s_{18} + s_{09} + s_{19})/4\). A new tuple of
\{n_{00}, \ldots , n_{04}\} will be emitted at the end. As next Window_A arrives, the Resize operator
will repeat the same operation on it.

(4) Here, another Aggregate is used to make a sliding window of size \(w = 3\) with a sliding factor
of 1, called Sliding_Window_B. It is created by accumulating 3 tuples of the incoming stream.
As a new tuple arrives, the oldest tuple get discarded. Every time a new tuple arrives, the
aggregation operation is computed and a tuple with the aggregated results is emitted.
For instance, in Figure 2 Subfigure (4), the contents in the current Sliding_Window_B are
\{n_{00}, \ldots , n_{04}; n_{10}, \ldots , n_{14}; n_{20}, \ldots , n_{24}\}. As a new tuple \{n_{30}, \ldots , n_{34}\} arrives, a new
Sliding_Window_B will have \{n_{10}, \ldots , n_{14}; n_{20}, \ldots , n_{24}; n_{30}, \ldots , n_{34}\}.

(5) Every Sliding_Window_B is passed to ComputeRatio, a UDOP. The operator ComputeRatio
has at hand a mask \(M\), which is a sliding window of size \(c/e\), called
a Stream_r tuple, except for the first and last Sliding_Window_B’s which generate two
Stream_r tuples. Each of these output tuples is computed according to Formula (1).

In the example shown in Figure 2 Subfigure (5), where the order of the mask (i.e. the
square matrix encoding it) is \(w = 3\), one can see the first Sliding_Window_B. The first Stream_r
tuple \{r_{00}, \ldots , r_{04}\} is computed from the incoming tuples \{n_{00}, \ldots , n_{04}\}
and \{n_{10}, \ldots , n_{14}\}. Subfigure (5) also shows the view when Formula (1) is applied
to compute \(r_{04}\). First, we locate the center of \(M\), \(n_{11}\), on top of \(n_{04}\). Then, by
Formula (1), \(r_{04}\) is the minimum ratio among its neighbors w.r.t the mask \(M\), that is,
\[r = \min \{n_{00}/m_{10}, n_{04}/m_{11}, n_{13}/m_{20}, n_{14}/m_{21}\}\]. Likewise, the second Stream_r tuple

(9) Subtract: subtract the background from the source image pixel-wise and tuple by tuple, e.g.
\(u_{00} = s_{00} - b_{00}, u_{01} = s_{01} - b_{00}\). Output tuples of pixel values freed of background.

Figure 2: Stream processing model of Algorithm 1 (Subfigure (9)).
\{r_{10}, \ldots, r_{14}\} \text{ is computed from the three incoming tuples in the same } \text{Sliding Window}_B.

Recall that the next \text{Sliding Window}_B is obtained by discarding the tuple \{n_{00}, \ldots, n_{04}\} and adding the tuple \{n_{30}, \ldots, n_{34}\}. In this case, the three incoming tuples are used to generate \{r_{20}, \ldots, r_{24}\}. More generally, and except for the last one, the three incoming tuples of \text{Sliding Window}_B are used to produce one \text{Stream}_r tuple. Finally, the last \text{Sliding Window}_B produces two \text{Stream}_r tuples, similarly to the first \text{Sliding Window}_B. More precisely, in this latter case, the three incoming tuples generate one \text{Stream}_r tuple; then the last two incoming tuples generate the last \text{Stream}_r tuple.

(6) A \text{Sliding Window}_C of size 3 with a sliding factor of 1 is created from \text{Stream}_r in the same manner as in Step (4).

(7) We develop a UDOP \texttt{EstimateBackground} with the same mask \(M\) as in Step (5) to compute the background values of the pixels from each \text{Sliding Window}_C according to Formula (2). \text{Sliding Window}_C’s with \text{Stream}_r tuples are used in the same manner as \text{Sliding Window}_B’s with \text{Stream}_n tuples are used in Step (5). After the background values of an incoming tuple are computed, for instance in Figure 2 Subfigure (7), the values of \(b_{00}, b_{01}, b_{02}, b_{03}, b_{04}\) for the first \text{Stream}_r tuple, we scale them up by a factor of 2 resulting in two output tuples of size 10, each containing \(\{b_{00}, b_{00}, b_{01}, b_{01}, b_{02}, b_{02}, b_{03}, b_{03}, b_{04}, b_{04}\}\). Thus, the operator \texttt{EstimateBackground} generates a sequence of tuples for the background values, called \text{Stream}_b.

(8) In Figure 2 Subfigure (8), a \texttt{Barrier} is used to synchronize and bundle \text{Stream}_s and \text{Stream}_b in order to match the original pixel values with their corresponding background values. This produces a stream where each tuple consists of two fields, one for a row of original pixel values and one for their corresponding background values.

(9) The above synchronized stream is passed to a UDOP \texttt{Subtract} to subtract the background from the original values pixel-wise for each tuple, producing a stream of data freed of background noise.

To this point, the stream of XRD data freed of background can be either passed to other operators in the same stream processing system for further analysis such as blob searching and peak fitting or output to an external device such as a file by the built-in \texttt{Sink} operator as in Algorithm 2 in Appendix. Overall, all the operators run concurrently on the elements in streams of data, until the end of the XRD source data. Since our pipelining implementation executes in the same computational order as the sequential algorithm, its correctness follows. It is noticeable that in this design there are two expensive operators \texttt{ComputeRatio} and \texttt{EstimateBackground} which workloads are rather balanced.

5. Experimentation

We have InfoSphere Streams Version 1.0 installed and configured on an Intel Core2 Quad CPU Q9550 (2.83 GHz, 8 GB RAM and 6 MB L2 cache) running with Red Hat Enterprise Linux 5.3 operating system. Given \(P\) image files, our program is parameterized so that it can set up \(P\) pipelines to process them. The \(P\) pipelines share the computing resources. Streams runtime system schedules the jobs and manages the resources, which is transparent to a user’s application.

The XRD source files in our tests are from a set of synchrotron experimental results in [3]. The size of each 2D XRD source file is 8 MB. It encodes a row-major layout matrix of 2084 \times 2084 pixel values in binary format. We set the resizing factor \(e\) to 2. The larger the resizing factor, the less the accuracy of the estimated background. According to our analysis in Section 3, the larger the filter size \(w\), the higher the computation cost. Moreover, what values of \(w\) that are
suitable depend on the type of X-ray diffraction patterns. As reported in [2] and [6], the Full Width at Half Maximum (FWHM) of a Bragg’s peak is in general between 10 and 40. We have used three different sizes of parabolic filters in our experimentation: 25, 45 and 85, which correspond to FWHM of values 10, 20 and 40. This allows us to test the performance of our pipelining implementation for different sizes of problems. Meanwhile it provides information on the effect of using different sizes of filters to help with users’ analysis. In Appendix, the sub-figures in Figure 4 show an XRD source image and the output images with the background removed by parabolic filters of size 25, 45 and 85, respectively.

Table 1: Timing and speedup for processing P files using P pipelines on a quad-core machine.

<table>
<thead>
<tr>
<th>P</th>
<th>Seq. (s)</th>
<th>Pipe. (s)</th>
<th>Sp.</th>
<th>Seq. (s)</th>
<th>Pipe. (s)</th>
<th>Sp.</th>
<th>Seq. (s)</th>
<th>Pipe. (s)</th>
<th>Sp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.93</td>
<td>2.82</td>
<td>1.7</td>
<td>16.1</td>
<td>8.4</td>
<td>1.9</td>
<td>55.8</td>
<td>26.4</td>
<td>2.1</td>
</tr>
<tr>
<td>2</td>
<td>9.86</td>
<td>3.18</td>
<td>3.1</td>
<td>32.2</td>
<td>9.0</td>
<td>3.6</td>
<td>111.6</td>
<td>27.5</td>
<td>4.1</td>
</tr>
<tr>
<td>3</td>
<td>14.8</td>
<td>4.81</td>
<td>3.1</td>
<td>48.3</td>
<td>10.8</td>
<td>4.5</td>
<td>167.4</td>
<td>35.8</td>
<td>4.7</td>
</tr>
<tr>
<td>4</td>
<td>19.7</td>
<td>5.78</td>
<td>3.4</td>
<td>64.4</td>
<td>14.5</td>
<td>4.4</td>
<td>223.2</td>
<td>45.4</td>
<td>4.9</td>
</tr>
<tr>
<td>5</td>
<td>24.7</td>
<td>6.45</td>
<td>3.8</td>
<td>80.5</td>
<td>18.0</td>
<td>4.5</td>
<td>279.0</td>
<td>57.3</td>
<td>4.9</td>
</tr>
<tr>
<td>6</td>
<td>29.6</td>
<td>8.11</td>
<td>3.6</td>
<td>96.6</td>
<td>21.5</td>
<td>4.5</td>
<td>334.8</td>
<td>66.1</td>
<td>5.1</td>
</tr>
<tr>
<td>8</td>
<td>39.4</td>
<td>10.1</td>
<td>3.9</td>
<td>128.8</td>
<td>28.6</td>
<td>4.5</td>
<td>444.6</td>
<td>93.9</td>
<td>4.8</td>
</tr>
<tr>
<td>10</td>
<td>49.3</td>
<td>13.3</td>
<td>3.7</td>
<td>161.0</td>
<td>36.2</td>
<td>4.4</td>
<td>558.0</td>
<td>112.2</td>
<td>5.0</td>
</tr>
<tr>
<td>12</td>
<td>59.2</td>
<td>15.6</td>
<td>3.8</td>
<td>193.2</td>
<td>42.7</td>
<td>4.5</td>
<td>669.6</td>
<td>135.6</td>
<td>4.9</td>
</tr>
<tr>
<td>14</td>
<td>69.0</td>
<td>18.2</td>
<td>3.8</td>
<td>259.2</td>
<td>49.4</td>
<td>5.2</td>
<td>781.2</td>
<td>158.6</td>
<td>4.9</td>
</tr>
<tr>
<td>16</td>
<td>78.9</td>
<td>20.8</td>
<td>3.8</td>
<td>257.6</td>
<td>56.4</td>
<td>4.6</td>
<td>892.8</td>
<td>181.9</td>
<td>4.9</td>
</tr>
</tbody>
</table>

Figure 3: Processing 4 XRD images using 4 pipelines in InfoSphere Streams.

We have obtained timings for processing P files using 3 different sizes of filters by P concurrent pipelines with P from 1 to 16 on the quad-core machine. Figure 3 is the Streams Live Graph captured by Streams Studio for processing 4 XRD source files by 4 pipelines concurrently. The timing results and the speedup factors are summarized in Table 1. Seq. denotes the sequential program. Pipe. is our pipelining implementation. The speedup (Sp.) factor of the pipelining program is computed w.r.t the timing for processing the same number of files by the sequential C program.

The timing for processing a XRD image by our program increases w.r.t the size of the filter. The sequential program takes about 5 seconds using a filter of size 25, 16 seconds using one of size 45 and about 55 seconds using one of size 85. By examining the quality of the output images users can choose a proper size, as the example shown in Figure 4 in Appendix.
We have developed a tool for measuring the timing of concurrent executions in Streams, as follows. For each pipeline, we create an auxiliary stream and use the Sink operator of SPADE to write the time for two special tuples into two separated files. One is the time when the first tuple from the source file is produced, called \textit{starting time}. The other one is the time when the last tuple of the result stream is emitted, called \textit{ending time}. For processing $P$ files using $P$ pipelines, there are $P$ records of starting time and $P$ records of ending time. We define the difference between the latest ending time and the earliest starting time to be the timing for a concurrent execution.

The speedup for processing one image using one pipeline is from 1.7 for a small-size problem to 2.1 for a large-size problem. The speedup of using one pipeline is obtained by the two costly but balanced operators, \textit{ComputeRatio} and \textit{EstimateBackground}. This result agrees well with our analysis in Section 3 and 4. To process 2 files using 2 pipelines, the parallelism is improved by the concurrent execution of two pipelines. We gain a speedup factor of 3.1 for the small-size problem, and 4.1 for the large one. The cases for processing many files (3 to 16) using the same number of pipelines create sufficient parallelism and potentially hide memory latency. They have achieved super-linear speedup on the quad-core machine.

In summary, our pipelining implementation performs well for all sizes of the problems that we have tested, and even better for the large one. It helps to improve the throughput of our XRD data processing by an order of more than the number of cores on a quad-core machine. It demonstrates high potential of making good use of multi-cores for processing large volume of XRD image data in InfoSphere Streams.

6. Concluding Remarks

Our initial work on stream processing of XRD images has been encouraging. While full analysis of individual XRD images requires further computation, we feel that the approach will result in a useful stream processing model for these images. Our overall objective was to develop analysis of XRD images where images acquired from a detector could be sent for analysis and results sent to the researcher nearly as quickly as they could be pulled from the detector. Our preliminary work suggests that the streaming approach and the use of the \textit{InfoSphere Streams} platform, has the potential to achieve this goal.

Further extension of this work include achieving performance enhancement at two levels. The first level is at the implementation details of the main algorithms. In particular, developing more fined-grained user defined operators to allow flexibility in paralleling lower steps. The second level is realizing faster file loading mechanism. Image files’ reading performances are bounded by the underlying file system throughput. Developing faster file loading techniques, will achieve gain in performance at the pipelining level and the algorithmic level.

In long term, this work provides the seed for the development of more general library of operators for processing images. In particular, operators, such as the background removal algorithm illustrated in this paper, could form part of a more general library of streaming operators for processing images, derived from a varieties of detectors. This will open a framework of collaboration among scientists around streaming technology and reusable assets.

7. Acknowledgments

We would like to acknowledge the support for this project from CANARIE and for the networking infrastructure that they provide to enable us to even begin to think about streaming image data. We would also like to thanks all those involved in the Science Studio project.
References


[8] IBM 2010 IBM InfoSphere Streams Version 1.0.10 Installation and Administration Guide


[10] Interactive Data Language (IDL) website: http://www.ittvis.com/ProductServices/IDL.aspx


[12] Sherry N 2009 C Program in Science Studio Project (University of Western Ontario, London ON, Canada)


[14] Tamura N 2010 Fortran Procedure "backfit_bruckner" in XMAS (the Advanced Light Source, Lawrence Berkeley National Laboratory, USA)


8. Appendix
Algorithm 2: Stream processing model of Algorithm 1 in SPADE

**Input**: file $F$ encoding a 2D XRD image $S$ of dimension sizes $r \times c$, a resizing factor $e$, a filter size $w$

**Output**: image $S$ freed of background noise

1. stream $s$ ([stuple : DoubleList])
   ::= Source () ["file : //F", udfinformat= "Parse"] { };

2. stream $\text{Window}_A$ ([Atuple : DoubleList])
   ::= Aggregate (s ⟨count(e)⟩) [ ] {Col(stuple)};

3. stream $\text{n}$ ([ntuple : DoubleList])
   ::= Udop (Window_A) ["Resize"] { };

4. stream $\text{Sliding\_Window}_B$ ([Btuple : DoubleList])
   ::= Aggregate (n ⟨count(w), count(1)⟩) [ ] {Col(ntuple)};

5. stream $\text{r}$ ([rtuple : DoubleList])
   ::= Udop (Sliding_Window_B) ["ComputeRatio"] { };

6. stream $\text{Sliding\_Window}_C$ ([Ctuple : DoubleList])
   ::= Aggregate (r ⟨count(w), count(1)⟩) [ ] {Col(rtuple)};

7. stream $\text{b}$ ([btuple : DoubleList])
   ::= Udop (Sliding_Window_C) ["EstimateBackground"] { };

8. stream $\text{sb}$ ([stuple : DoubleList, btuple : DoubleList])
   ::= Barrier (s; b) [ ] {$1.\text{stuple}, 2.\text{btuple}$};

9. stream $\text{u}$ ([utuple : DoubleList])
   ::= Udop (sb) ["Subtract"] { };

10. $\text{Nil}$ ::= Sink (u) ["file : //F.out"] { };}
Figure 4: An XRD image example.