INTRODUCTION TO INTEL DAAL, PART 2: DISTRIBUTED VARIANCE-COVARIANCE MATRIX COMPUTATION

Ryo Asai

Colfax International

March 28, 2016

Abstract

This is part 2 of 3 of an introductory series of publications on the Intel Data Analytics Acceleration Library (DAAL). In part 1 of the series we discussed how to implement batch mode computation on a single node. In the present publication, we discuss the distributed mode computation.

Our discussion will focus both on how and when to implement distributed mode computation with Intel DAAL. As an example workload, we implement an application that uses DAAL to compute a covariance matrix of a set of vectors. We first demonstrate how to use distributed mode with this example. Then, using this example application, we scan the parameter space to determine what parameter ranges benefit from distributed computation. We also demonstrate how the output of this computation may be used in image processing to compute the eigenvectors of a set of images. The source code for this application is available for free download.

In the upcoming 3rd part of the series we will discuss the online computation mode, using an example workload with multiple datasets and interfacing with a relational database via SQL.

Table of Contents

1 Introduction **2**

2 Distributed Mode Computation **3**
2.1 Step-1: Local 4
2.2 Step-2: Master 5

3 Communication and Data **6**
3.1 Serialization and Deserialization 6

4 Example Implementation **8**
4.1 Workflow 8
4.2 Implementation 9

5 Example Use Case: PCA and Eigenimages **11**
5.1 Implementation 11
5.2 Result 12

6 Closing Words **13**

A Performance Considerations **14**
A.1 System Configuration 14
A.2 Tuning Single Node Performance . . . 14
A.3 Predicting Performance on the Cluster . 15

B Miscellaneous **19**
B.1 MPI Pinning 19
B.2 Amdahl's Law 19

1. Introduction

This is the second part of a series of Colfax Research publications introducing the Intel Data Analytics Acceleration Library (DAAL). In part 1 of the series [2] we discussed the batch mode, where DAAL computation proceeds on a single node. In this part, we discuss the distributed mode, where DAAL computation is processed on multiple compute nodes.

It is important to note that in its current implementation, Intel DAAL does not have built-in communication between distributed memory systems. Instead, DAAL provides the tools for data management and division of workload that enables distributed computation. Therefore the communication aspect of the distributed computation must be implemented by the developer. Any API that is capable of transferring data packets across distributed memory systems may be used (e.g., Hadoop). In this publication we are using a framework called Message Passing Interface (MPI) for this purpose, which is an industry-standard tool in high performance computing.

Figure 1 illustrates the workload discussed in this paper.

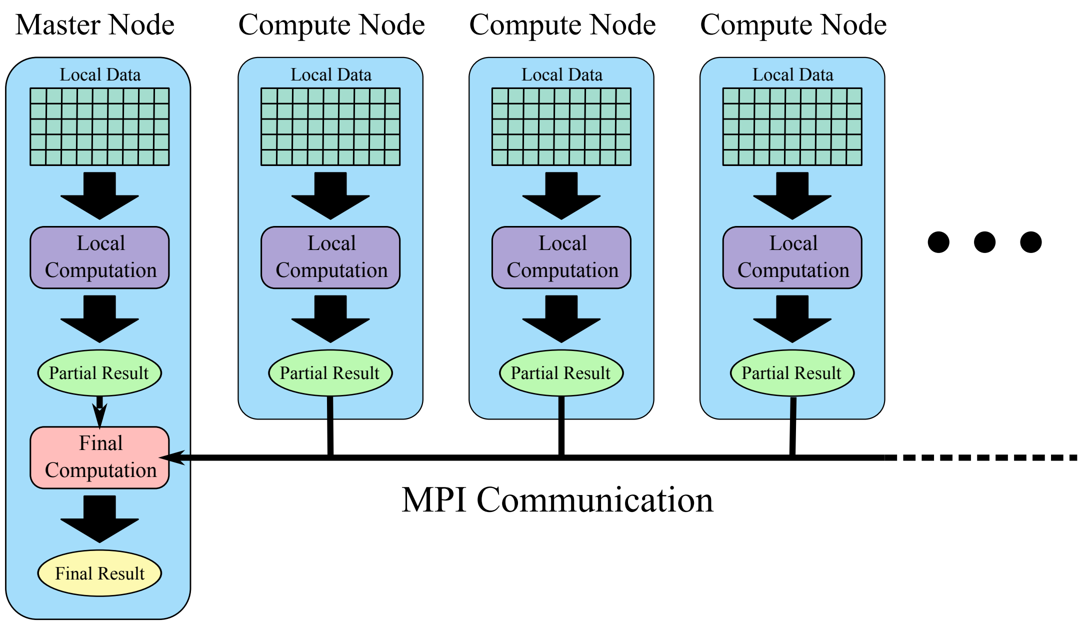

Figure 1: Distributed computation with DAAL.

In Section 2 we demonstrate how to use the computation aspect of the distributed mode. Then, in Section 3, we discuss the data management and manipulation aspects required for the communication. Finally, in Section4 and Section 5, we combine the two and implement Variance-Covariance Matrix computation for a set of vectors, illustrating a potential application with an image processing application.

Extended Appendix A and Appendix B report on distributed performance measurements. There we study the effect of the size of the problem set on applicability of using the distributed mode on a cluster.

The result of our work is available for free download at [1].

2. Distributed Mode Computation

With most Intel DAAL algorithms, the typical workflow of distributed computation is in two steps:

Step 1: **local computation**. Using a part of the dataset, or the entire dataset, each compute node calculates a partial result.

Step 2: **master computation**. The partial produced in Step 1 results are gathered in one compute node (master) and combined into a final result.

There are few exceptions to this two-step flow (e.g. QR decomposition where more steps are involved), but distributed algorithms with more steps also have both "local computation" portions and "master computation" portions.

Because there are two separate steps, local and master, a separate algorithm object must be created for each step. Thus, unlike the batch mode algorithm object (see [2]), the distributed mode algorithm has the option `required` for selecting which step this object is used for. For example, with the covariance algorithm, the two arguments are:

```
covariance::Distributed<step1Local>  algorithmLocal;
covariance::Distributed<step2Master> algorithmMaster;
```

The `step1Local` algorithm object must be created for each compute node, but the `step2Master` algorithm object only needs to be created on one of the compute nodes.

In this section, we will use covariance algorithm for demonstration of distributed computation. Furthermore we will be working in the following namespaces in this section for convenience.

```
using namespace daal;
using namespace daal::data_management;
using namespace daal::algorithms;
```

2.1. Step-1: Local

Listing 1 shows local computation step of the covariance algorithm in the distributed mode.

```
// Populating Numeric table (discussed in the first paper)
services::SharedPtr<NumericTable> dataMatrix(PartOfDataNumTable);

// Creating local algorithm and setting the inputs
covariance::Distributed<step1Local> localAlgorithm;
localAlgorithm.input.set(covariance::data, dataMatrix);

// Computing the partial result
localAlgorithm.compute();

//Extracting the partial result, and sending it to master
services::SharedPtr<covariance::PartialResult> partResult;
partResult = localAlgorithm.getPartialResult();
SendPartResultToMaster(partResult);
```

Listing 1: Example local computation implementation.

In order to benefit from distributed computing resources, each local computation must process only a part of the original dataset. Therefore, a *partial dataset for the local compute node* must be created. This partial dataset is represented by `PartOfDataNumTable` in Listing 1. For discussion on numeric tables and loading data, refer to the first paper [2].

Next, we must create a local algorithm object with the `step1Local` option, then compute the partial result from the partial dataset. The input must be set using the `input.set()` method of the algorithm object, After that, the `compute()` method must be used to run the computation. Finally, the local result, stored in a `partialResult` object, can be extracted with the `getPartialResult()` method. Note that with the *local* algorithm, only one set of data is allowed per algorithm object. To work with more datasets in one compute node, simply create another algorithm object and repeat the procedure.

The partial result must now be sent to the host. The communication part is discussed in Section 3.

2.2. Step-2: Master

Listing 2 shows master computation step of the covariance algorithm in the distributed mode.

```
// Creating master algorithm
covariance::Distributed<step2Master> masterAlgorithm;

// Multiple partial results being combined
services::SharedPtr<covariance::PartialResult> partResult[NumberOfPartResults];
for(int ptResult =0; ptResult < NumberOfPartResults; ptResult++) {
  partResult[ptResult] = getNextPartResult();

  // Appending the next partial result;
  masterAlgorithm.input.add(covariance::partialResults, partResult[ptResult]);
}

// Combining results into one to get the final result
masterAlgorithm.compute();
masterAlgorithm.finalizeCompute();

// Extracting the final result
services::SharedPtr<covariance::Result> covarianceResult;
covarianceResult = masterAlgorithm.getResult();
```

Listing 2: Example master computation implementation.

The workflow of the master step is similar to that of the local step.

We first create the master algorithm object with the `step2Master` option. The inputs of the master algorithms are partial results from various compute nodes. For the transfer of partial results, refer to Section 3. Partial results must be added to the master algorithm with the `input.add()` method of the master algorithm object. Note that here, unlike the local algorithm object, we add multiple partial results to the master algorithm.

When all the partial results are added, the partial results are combined with the `compute()` method and the result is calculated with `finalizeCompute()` method.

The result is stored in the `covariance::Result` object as a member `covariance::covariance`. The final lines in Listing 2 demonstrate how to extract the pointer to an array of elements of type `double` containing the result data. For more on the `BlockDescriptor<>`, refer to the first paper [2]. Finally, for more information on specific implementation for other algorithms, refer to the DAAL documentation [3].

3. Communication and Data

In this section we discuss the communication aspect of distributed computation. In our implementation of distributed computation (see Section 2) we use a framework called MPI for data transfer between distributed systems. The discussion on how to use MPI is outside the scope of this publication, however there are multiple MPI tutorials available (e.g., [4]).

In order to transfer data across distributed memory systems with MPI, we must use one of the MPI communication routines, such as `MPI_Send()` and `MPI_Recv()`. However, MPI routines can only transfer arrays of basic types; our `PartialResult` objects can not be transferred as is. To allow for easier transfer of objects in DAAL, the library has support for data serialization. With data serialization, objects are converted into a byte array, which can later be deserialized to recover the original object. This allows us to convert our `PartialResult` objects to a form that we can transfer with MPI.

In the remainder of the section we will demonstrate a serialization and deserialization example. We work in these namespaces in the code samples to follow.

```
using namespace std;
using namespace daal;
using namespace daal::data_management;
```

3.1. Serialization and Deserialization

Listing 3 shows the procedure for serialization of partial results and the data transfer using MPI.

```
// ... Local Computation ...
services::SharedPtr<covariance::PartialResult> partResult;
partResult = localAlgorithm.getPartialResult();

// Serialization
InputDataArchive partResultArchive;
covariancePartResult->serialize(partResultArchive);

// Finding the size of the serialized archive, and notifying the master node
size_t buffer_size = partResultArchive.getSizeOfArchive();
MPI_Send(&buffer_size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);

// Copying the data into a buffer to be sent to the master node
byte* send_buffer = (byte*) malloc(buffer_size);
partResultArchive.copyArchiveToArray(send_buffer, buffer_size);
MPI_Send(send_buffer, buffer_size, MPI_BYTE, 0, 0, MPI_COMM_WORLD);
```

Listing 3: Example serialization implementation.

With data serialization, the first step is to create an `InputDataArchive` object, which serves as the container for the serialized data. Then pass the container as an argument to the `serialize()` method to load the serialized data to the archive object. Note that the data serialization method, `serialize()`, is a method of the object to be serialized, not the archive object (see line 7). A subset of objects in Intel DAAL, including the PartialResult object, support this function and thus can be serialized. To determine whether a particular object is serializable, refer to the DAAL documentation.

The size of the byte array in the archive can be found with `getSizeOfArchive()` method. In our example, this size is used to create the buffer on both the master node and the local compute node. We then copy the contents of the archive object to a basic byte array using the `copyArchiveToArray()`. Although it is also possible to directly access the data in the archive, we copy it to a separate send buffer for clarity. The contents of the copied buffer are then sent to the master node using the `MPI_Send()` routine.

Listing 4 shows the procedure for deserialization of partial results and the data reception with MPI.

```
MPI_Status stat;

// Determining the size of the buffer for receiving data
int buffer_size;
MPI_Recv(&buffer_size, 1, MPI_INT, i, 0, MPI_COMM_WORLD, &stat);

// Receiving the serialized data
byte* recv_buffer = (byte*) malloc(buffer_size);
MPI_Recv(recv_buffer, buffer_size, MPI_BYTE, i, 0, MPI_COMM_WORLD, &stat);

OutputDataArchive receivedArchive(recv_buffer, buffer_size);
services::SharedPtr<covariance::PartialResult> receivedPartResult =
                           services::SharedPtr<covariance::PartialResult>(
                                            new covariance::PartialResult() );
receivedPartResult->deserialize(receivedArchive);
```

Listing 4: Example deserialization implementation.

The deserialization procedure is simply the reverse of the serialization procedure. For data deserialization, after receiving the byte array, create a `OutputDataArchive` object from the buffer that contains the byte array. Just like its input counterpart, this only serves as the container. Deserialization itself is carried out with the `deserialize()` method of the object to recover, and the archive is passed in as the argument to the deserialization function.

4. Example Implementation

For our example implementation, we will use distributed computation mode to find the covariance matrix of a large set of vectors. Covariance matrix measures how much two variables "vary" together. The input of the workload is a set of N_v vectors each with p elements. This data is represented as an $n \times p$ matrix. The result is a $p \times p$ Variance-Covariance matrix where the i,j-th element is the variance between the i-th and j-th variables in the vector.

4.1. Workflow

Figure 2 shows the workflow of the example application.

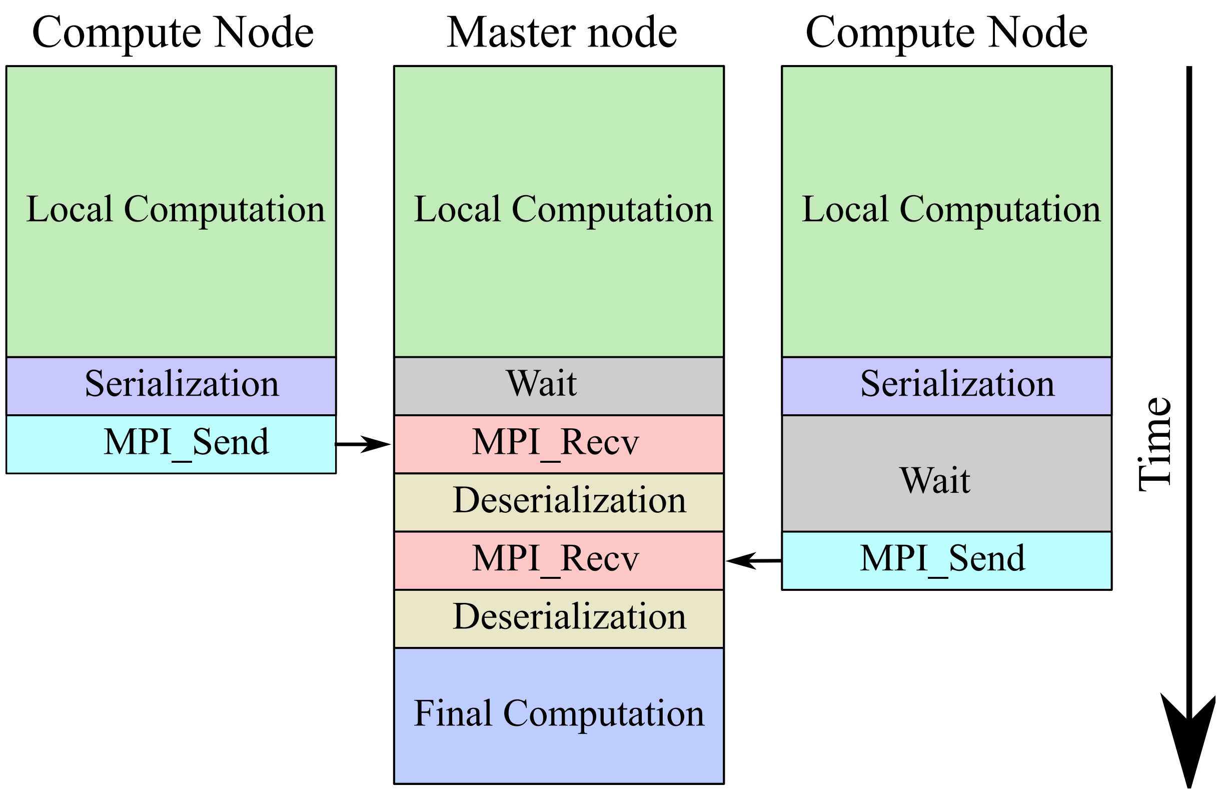

Figure 2: Workflow in time of the example application.

We assume that the dataset originally exists on every node. Each compute node as well as the master node will carry out local computation at the same time. Once the local computation is completed, the result object in each compute node, but not the master node, is serialized in order to prepare it for transfer.

We use basic point-to-point MPI communication to gather the partial results from compute nodes to the master node. After receiving each partial result, the master node deserializes it to convert it back to a result object. Then this result object is added to the master algorithm object. When all the partial results are added, the master compute node will compute the final result.

Our minimal implementation does not include load balancing across the compute nodes. Load balancing may be necessary, for example, if the dataset is not distributed evenly between the nodes. Boss-worker or work stealing models may be used for load balancing in MPI.

4.2. Implementation

We have implemented the simple distributed computation discussed above, and the source code is available for download from [1]. In the source code, as well as in this section, we will be using Message Passing Interface (MPI) v3.0 for communication. For information on MPI, refer to [4].

The application can be mostly split into two parts: the first part is the local computation, and the second part is the communication and master computation.

In local computation, all MPI processes independently compute the partial result from its portion of the dataset. This portion is near identical to the example in Section 2.1: refer to the section explanation of the implementation of this portion.

In the communication part, the application flow splits into two branches: master branch `rank==0` and worker branches `rank!=0`, where `rank` is the MPI process rank, i.e., its unique numerical identifier. The workers are responsible for serializing their partial results, and master is responsible deserializing these partial results and computing the final combined result.

Listing 5 shows the worker portion of the communication and serialization.

```
// Worker node just needs to package the partial result and send it off

// Serialization of partial results
InputDataArchive partResultArchive;
covariancePartResult->serialize(partResultArchive);
size_t buffer_size = partResultArchive.getSizeOfArchive();

// Copying the archive into the send buffer
byte part_buffer[buffer_size];
partResultArchive.copyArchiveToArray(part_buffer, buffer_size);

// Sending the partial result
MPI_Send(&buffer_size, 1, MPI_INTEGER8, 0, 0, MPI_COMM_WORLD);
MPI_Send(part_buffer, buffer_size, MPI_BYTE, 0, 0, MPI_COMM_WORLD);
```

Listing 5: Worker node serialization and communication.

The partial result is serialized using the `serialize()` method of the partial result. Then we first send the size of the buffer for two purposes: to notify the master node that partial result is ready to be transferred, and also to notify the master node what the required size of the receiving buffer is. Because MPI messages are non-overtaking, the master node is guaranteed to receive and process the buffer size prior to receiving the message with the buffer contents.

Listing 6 shows the master portion of the communication and deserialization.

```
//Creating master algorithm
covariance::Distributed<step2Master> algorithmMaster;

// Add the following line to produce a correlation matrix instead
// algorithmMaster.parameter.outputMatrixType = covariance::correlationMatrix;

// Adding Master's partial result
algorithmMaster.input.add(covariance::partialResults, covariancePartResult);

MPI_Status stat;
byte* part_buffer;
for(int i = 1; i < worldSize; i++) {
  // Receiving the serialized partial results
  size_t buffer_size;
  MPI_Recv(&buffer_size, 1, MPI_INTEGER8, MPI_ANY_SOURCE, 0,
                                     MPI_COMM_WORLD, &stat);
  int worker = stat.MPI_SOURCE;
  if(part_buffer == NULL) {
    // NOTE: If the partial results are large, allocate on heap instead
    part_buffer = new byte[buffer_size];
  }
  MPI_Recv(part_buffer, buffer_size, MPI_BYTE, worker, 0,
                                   MPI_COMM_WORLD, &stat);

  // Deserializing the received partial result
  OutputDataArchive receivedArchive(part_buffer, buffer_size);
  services::SharedPtr< covariance::PartialResult > receivedPartResult =
                        services::SharedPtr<covariance::PartialResult>(
                                          new covariance::PartialResult());
  receivedPartResult->deserialize(receivedArchive);

  // Adding the partial result that was received
  algorithmMaster.input.add(covariance::partialResults, receivedPartResult);
}
```

Listing 6: Master node deserialization and communication

We first create the master algorithm and add the master's partial result. Then we begin listening for notification from other nodes using `MPI_Recv`. Here we use `MPI_ANY_SOURCE` for the message source so that we can receive the message from the first node that sends `buffer_size`.

Once the master process receives the `buffer_size`, it allocates the receive buffer. However, note that in our implementation, we only allocate on the first message we receive. This is because in the case of the covariance matrices, all partial results have the same size regardless of how many vectors were used to compute it. Thus, in order to reduce the time spent on allocating the buffer, we only allocate on the first message. But note that in some other algorithms, the partial result may vary from one compute node to another. The master process then receives the partial result *from the node that sent the* `buffer_size` message. We ensure this by querying the sender's rank of the first message, and setting the source of the second `MPI_Recv` to this rank. Finally, the master process deserializes the partial result that it received and adds it to the master algorithm object.

Once all the partial results are accounted for, the master node carries out the master computation to produce the final result. For more on the final computation step, see Section 2.2. The resultant covariance matrix is stored in `Result` object. The procedure for extracting the Numeric table and the raw pointer from algorithm results is discussed in the first paper [2].

5. Example Use Case: PCA and Eigenimages

Covariance matrices and their derivative, correlation matrices, have applications in variety of data analytics domains. One of the use cases for correlation matrix in conjunction with another method, called Principle Component Analysis (PCA). Given a large set of input vectors, PCA produces a shorter list of mutually orthogonal vectors (eigenvectors), linear combinations of which be used to approximate the original dataset with minimal loss of information. PCA takes as input the correlation matrix of the dataset, and as output it returns a set of eigenvectors sorted by their corresponding eigenvalues. This allows the user to pick a small subset of the most information-rich eigenvectors. This subset is called principle components, and its size is determined by the tolerance of the user's next stage of data processing to the accuracy of this approximation. PCA may help to significantly reduce the dimensionality of the dataset, which can allow for more accurate analysis of a large dataset with other techniques, such as classification algorithms or neural networks.

In the domain of image analysis, this techniques allows to approximate images in a large dataset with a superposition of images from a much smaller dataset. Figure 5 demonstrates this idea.

The general procedure for generating principle components of an image dataset is as follows. First, the 2D images are loaded and represented as 1D vectors. Typically, some pre-processing techniques are applied to the images to improve the quality of the input images. Then, the correlation matrix is computed from this large set of vectors (images). This matrix contains the correlation between the intensity values of all pairs of pixels within the image canvas. After that, a PCA procedure is applied to correlation matrix. It outputs orthonormal eigenvectors of the correlation matrix sorted sorted by their eigenvalues. The details of the mathematics behind the eigenimage production is beyond the scope of the publication: however, there are variety of documents regarding this topic, including [5].

The principal component eigenimage projection can be used in image processing and analysis problems, like face recognition and optical character recognition (OCR). In these applications, sometimes the number of images to be analyzed can be large. Using the distributed computation of correlation matrix with DAAL, this workload can be distributed among multiple compute nodes to expedite the analysis procedure.

In the next section, we will discuss how image reconstruction in 5 was implemented using DAAL.

5.1. Implementation

In our implementation we do not do any pre-processing of the images beyond subtracting the mean-value image. The mean-value image is the image where each pixel is the average pixel value across all images. This mean-value will be needed later when reconstructing images from eigenimages. Our dataset contains 20 square 60 px $\times$ 60 px images of the author. Thus, each image vector is $60 \times 60 = 3600$ elements long.

After subtracting the mean image, we use the DAAL correlation algorithm in distributed mode to compute the correlation of the pixel values across all images. The output correlation matrix can be fed directly into the DAAL PCA algorithm, using the correlation mode for PCA algorithm.

PCA algorithm outputs the orthonormal eigenvectors and the associated eigenvalues of the correlation matrix. These output eigenvectors are pre-sorted by the eigenvalues. Using the eigenvalues we choose the most information-rich eigenimages such that the sum of their eigenvalues is equal to 90% of the sum of all eigenvalues. This corresponds to preserving 90% of variance associated with the eigenimages [5]) in the basis of the new coordinates. For our problem set, the first 5 eigenimages accounted for 90% of the variance. Finally, we project the original image vectors to these eigenvectors to transform the images into the eigenimage space.

5.2. Result

Figure 3 shows several of the 20 images used in the application. Figure 4 shows the eigenimages that were produced through the procedure described in the previous section.

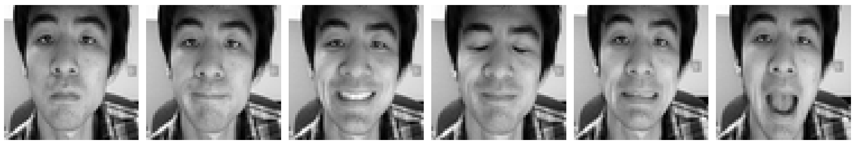

Figure 3: A subset of the original images.

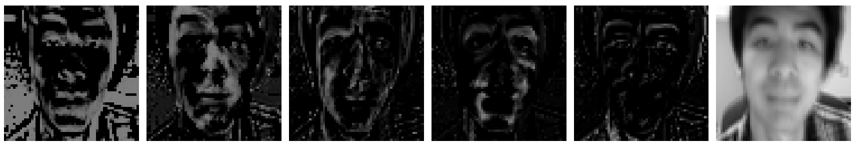

Figure 4: The six eigenimages and the mean-value image.

Finally, Listing 7 shows the coordinates of the images in Figure 4 in the eigenimage space.

```
 30.5784  1233.66 -570.971  -185.12  262.222
 552.564  564.893 -177.948 -90.8299 -429.371
 998.585 -250.623  54.7034 -341.388  370.104
-792.024   246.01  294.965  -226.33  1.80345
-956.119  284.556  488.508  69.6365  132.556
```

Listing 7: The eigenspace representation of the images. The top row corresponds to the left-most picture in Figure 4, second row to the second picture and so on. The first value in each row is the contribution of the first eigenimage, the second is the second eigenimage, and so on.

In order to reconstruct an image from its eigenimage space representation to pixel representation, simply perform the reverse transform, then add the mean-value image. Figure 5 demonstrates the procedure for the first image in Listing 7, as well as the resultant reconstructed image.

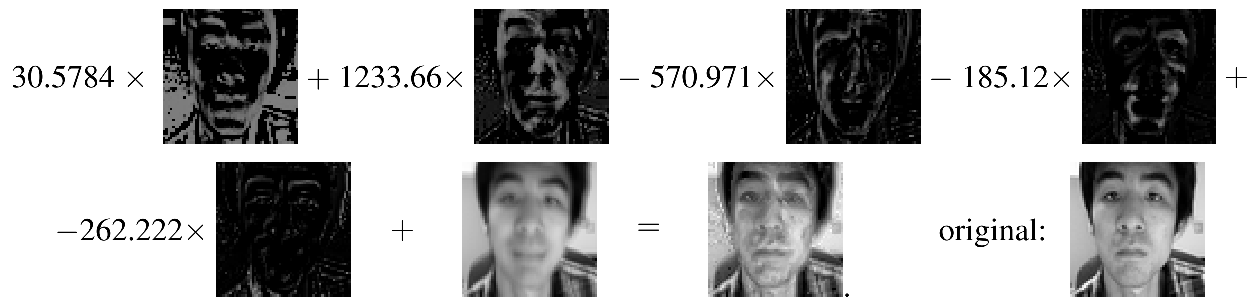

Figure 5: Reconstructing the images.

As can be seen from the reconstructed image, the 5 eigenimages and the norm-image are sufficient to make an approximation of the original image. If additional accuracy in the reconstruction is required, one may increase the percentage of the eigenvalue that is kept (e.g. instead of 90%, use 99%), producing a large set of eigenimages.

In our case, we only worked with 20 images. However, in cases where there are far more images to be decomposed, the distributed mode correlation computation from DAAL can scale this workload across multiple nodes.

6. Closing Words

This publication is the second part in a series of three Colfax Research publications introducing Intel DAAL. We focused on the distributed computation mode of the Intel Data Analysis Acceleration Library, and discussed the implementation of the two versions of the algorithm objects, `step1Local` and `step2Master`, as well as other tools necessary for distributed computing, like serialization and deserialization. In the next part of the series, we will visit one more mode of DAAL operation, the online computation mode, and discuss how to interface DAAL with a MySQL database. The source code for this series, as well as part 1 and 3 of the series, can be found at the Colfax Research website [1].

References

[1] Ryo Asai. Introduction to Intel DAAL, Part 2 of 3: Distributed Variance-Covariance Matrix Computation, 2016 (*landing page for this paper).*
http://colfaxresearch.com/intro-to-daal-2/.

[2] *Ryo Asai. Introduction to Intel DAAL, Part 1 of 3: Polynomial Regression with Batch Mode Computation, 2015.*
http://colfaxresearch.com/intro-to-daal-1/.

[3] *Download link for DAAL User and Reference Guide.*
https://software.intel.com/en-us/intel-daal-support/documentation.

[4] *Message Passing Interface Forum. MPI: A Message Passing Interface Standard Version 3.0.*
http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[5] *M. Turk and A. Pentland. Eigenfaces for Recognition.* Journal of cognitive Neuroscience*, 3:71–86, 1991.*

Appendix A. Performance Considerations

In the this publication, we discussed how to design an application that uses Intel DAAL in the distributed mode. In this section, we discuss how to *run* the application in a way that maximizes performance. We will first examine how the number of MPI processes per node affects performance, even on a single node. Then we will discuss the procedure for estimating performance in a cluster with arbitrary number of nodes. We use the correlation matrix computation as the primary example in this section, however, the ideas discussed in this section can be applied to any distributed computation with Intel DAAL.

We used simulated datasets with different numbers of vectors, N_v. By varying N_v, we can study the effect of the input dataset size on the applicability of the distributed computation.

A.1. System Configuration

All of the benchmarks presented in this section were taken on a cluster of four identical server systems with two-way Intel Xeon E5-2670 processors (16 cores per socket, 32 cores total in each server). For our tests, the Intel hyper-threading feature was enabled, and CPU clock frequencies were fixed at the maximum frequency of 2.60GHz. The communication fabric between the systems was with the Intel Omni Path Architecture, Alpha version.

We used the Intel C++ compiler version 16.0.1 and Intel DAAL 2016 Update 1 on CentOS 7.1 Linux OS.

A.2. Tuning Single Node Performance

Although the distributed computation mode is designed for scaling applications across distributed memory systems, it can also result in performance benefits even in a shared memory setting. In this portion we only look at how performance is affected by running multiple processes on the same system. For more discussion on running multiple processes on the same node and its benefits, refer to Appendix B.1.

Essential functions in Intel DAAL use the OpenMP parallel framework to scale across multiple cores. At the same time, threads used by DAAL respect the Intel MPI pinning scheme. Therefore, it is possible to utilize a multi-core compute node in a variety of ways:

1. One MPI process per node utilizing across all threads
2. Multiple MPI processes per node, each utilizing only 1 threads
3. Several multi-threaded MPI processes per node

The and the number of processes per node can affect performance considerably, especially in a compute node with a two-way Intel Xeon processor, because such a node is built as a Non-Uniform Memory Access (NUMA) system. Therefore, before the application is scaled across a cluster, it is important to investigate what the best number of processes per node is.

Figure 6 shows the performance of computing a 128x128 correlation matrix from N_v=65536 vectors as a function of the number of MPI processes per node. For each point, the product of the number of processes and the number of threads per process is equal to 32, the number of logical CPUs in the node. Performance is reported as number of vectors processed per second.

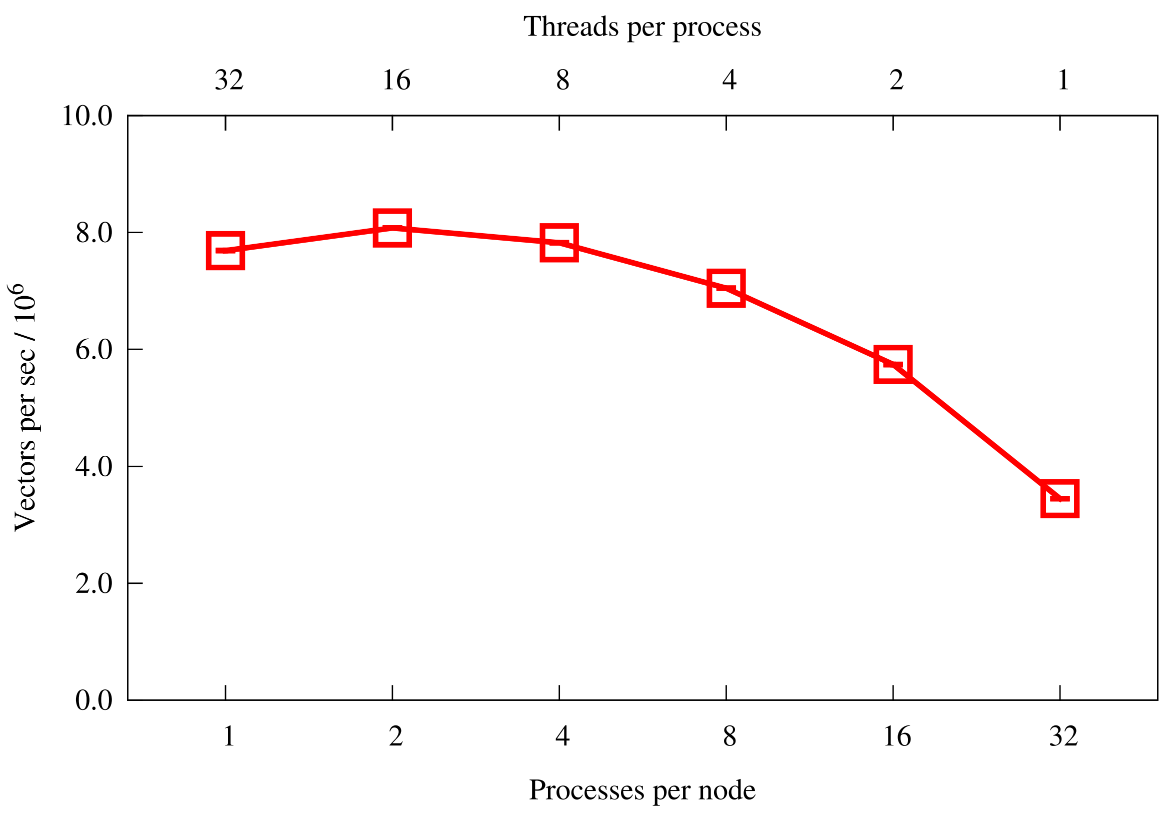

Figure 6: Performance as a function of processes per node.

From Figure 6 we can see that the best performance is achieved when we run two processes in the same node, each with 16 threads. This finding is specific to our system and application, however, it may be used as a starting point for in other systems and applications. The message here is not that you should run 2 processes per node, but that the number of processes per node is a tuning parameter that can greatly affect performance.

A.3. Predicting Performance on the Cluster

When developing an application for a large cluster, the developer may not have access to the full cluster but instead to just a couple of nodes. In such cases, the ability to estimate the performance on a large cluster based on data from a small one can be beneficial. In this section we discuss how to find the expected parallel efficiency of a distributed DAAL application on a large cluster, using data benchmarked on a smaller cluster. In this section, let us consider a scenario where the developer has access to two nodes initially.

First, we run the application across just two nodes in order to see the contribution of different components of the application (local computation, deserialization, etc.) in terms of execution time. Figure 7 shows the execution time of different parts of the application running on two nodes, as a function of the total number of vectors, N_v.

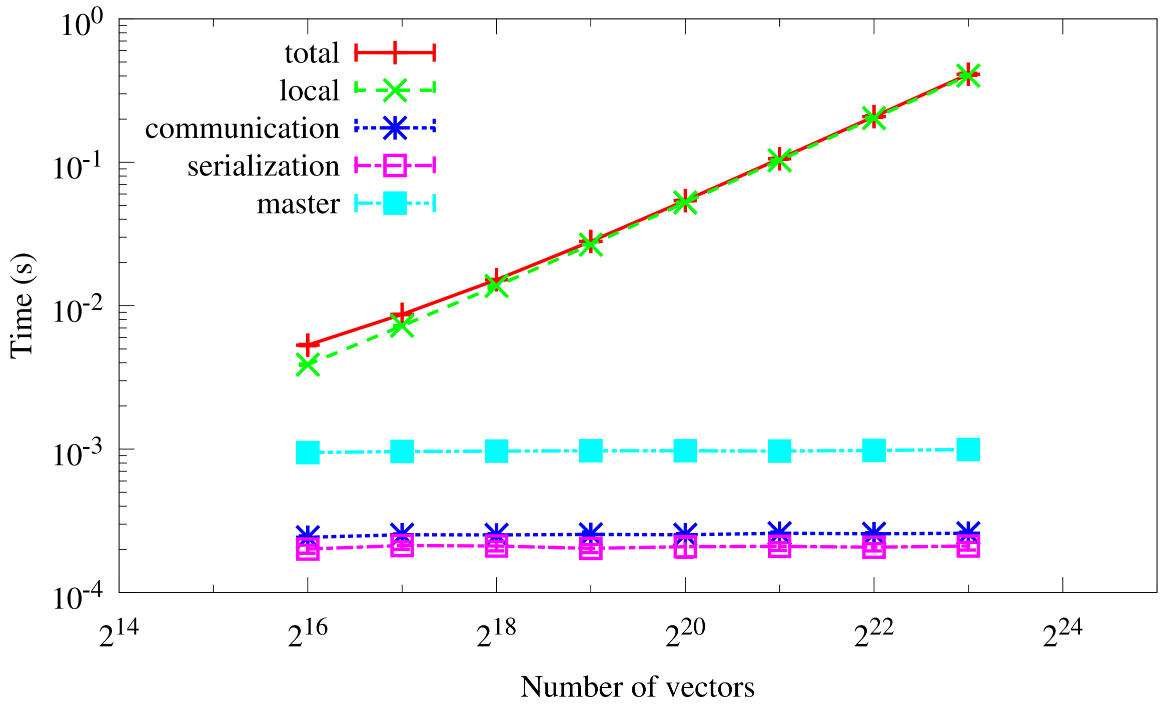

Figure 7: Contributions to the run time from different parts of the application.

In Figure 7, the data labelled "local" and "serialization" are the average execution times of *individual* local computation and serialization, respectively. On the other hand, "communication" and "master" are *total* execution times of communication and combination (deserialization plus master computation), respectively. This is because local computation and serialization can be done in parallel across multiple processes, whereas the communication and combination is limited by a single process `rank = 0`. Thus, although measured differently, the times reported in Figure 7 are the contributions of each part with respect to the total execution time of the whole application.

Figure 7 shows us that only the local computation scales with the number of vectors, N_v. This is the expected behavior because the serialization, communication, and combination all depend on the size of the partial result. The size of partial result depends on the size of the resultant correlation matrix (128x128 in this case), which is unaffected by N_v. For the local computation, we can see that it scales linearly with N_v.

We can now use this data in conjunction with Amdahl's law to approximate the expected parallel efficiency for different situations. After some algebra (see appendix B.2) we arrive at the equation (1):

$$E(p,s) = \frac{S(p,s)}{s} = \frac{1}{s(1-p)+p}, \quad \text{where} \quad p(N_v,s) = \frac{c_1 N_v + t_{\text{ser}}}{c_1 N_v + t_{\text{ser}} + c_2 s}. \tag{1}$$

Here, N_v is the number of vectors , and s is the number of nodes. Using the data attained from 7, we estimated c_1 and c_2 specific to our system and our application.

Let us now test equation (1). We scaled the correlation matrix application across four nodes. By substituting $s = 4$, we can get the parallel efficiency as a function of N_v. Figure 8 shows the *measured* parallel efficiency of the application as a function of N_v, as well as the predicted value.

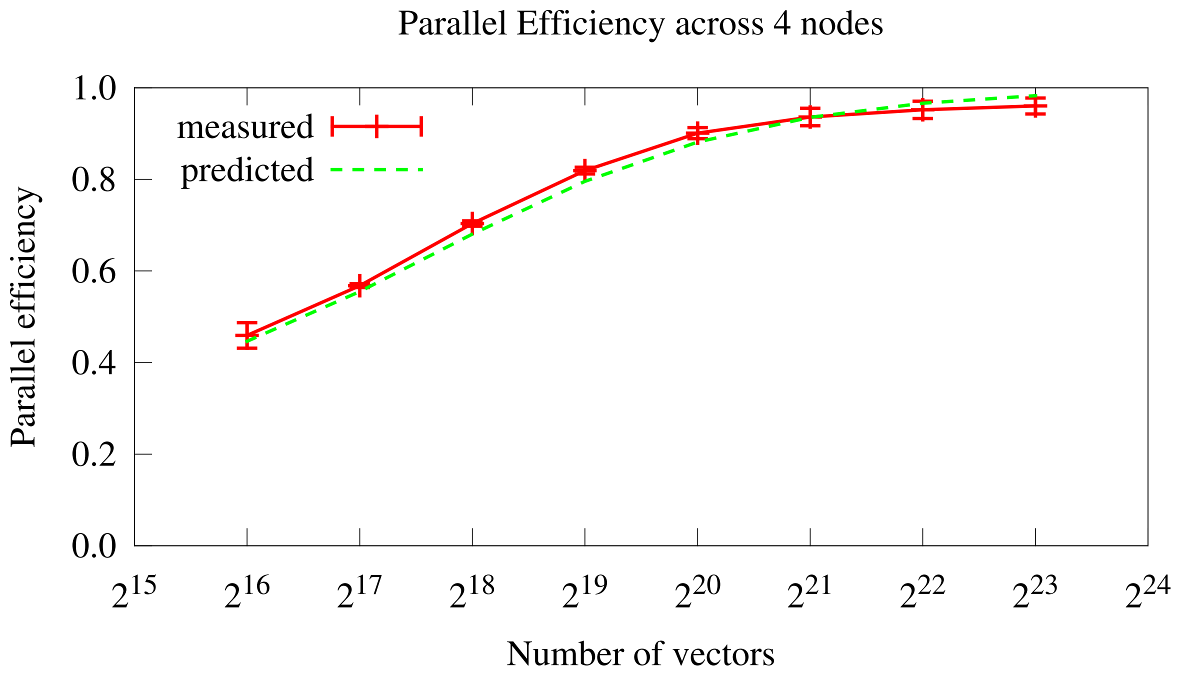

Figure 8: Measured and predicted parallel efficiency of the DAAL application.

The estimated parallel efficiency is close to the actual measured value. Such a graph can be used to determine the minimum size of the workload required to efficiently take advantage of a cluster. For example, Figure 8 tells us that to get 90% efficiency of the 4-node cluster, you would need at least $N_v = 2^{20}$ vectors. The same procedure can be applied for an arbitrary s to get a rough estimate on a larger cluster. (*Note:* the accuracy of this method typically decreases with the size of the cluster)

However, typically the question is not "given s, what is the minimum N_v" but the opposite. Given a fixed problem size (N_v), equation (1) can also be used to develop a model that predicts the expected speed-up from additional compute nodes. Figure 9 shows an example of the expected speed-up graph for several different values of N_v.

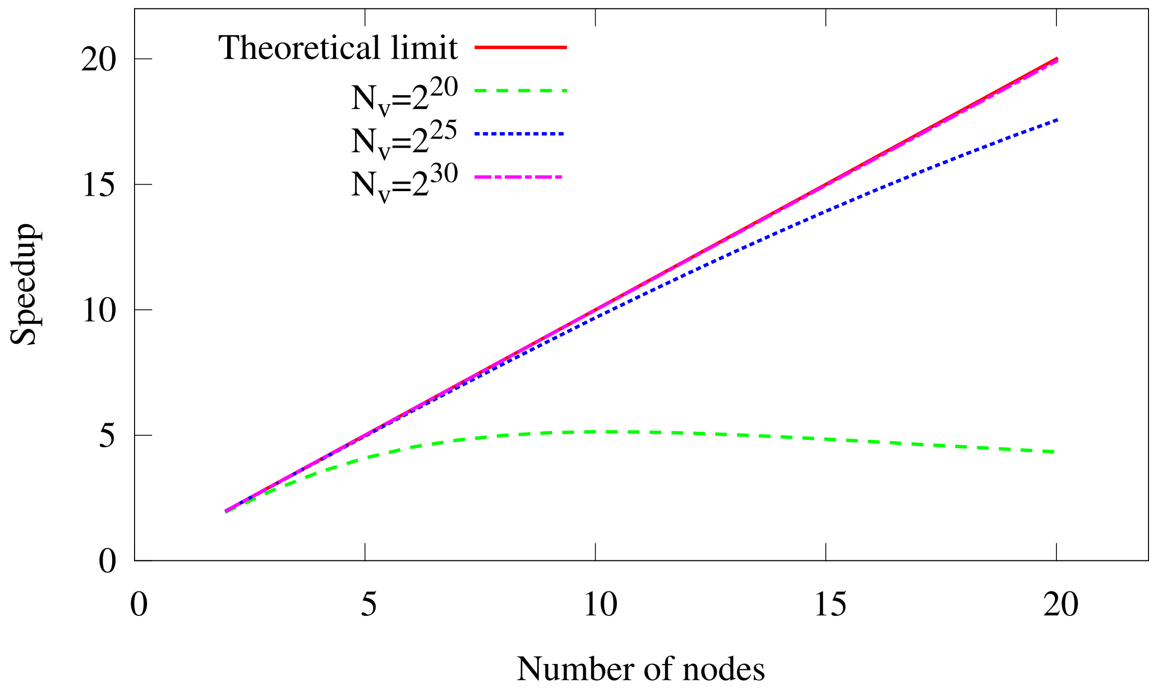

Figure 9: Predicted speedup versus number of nodes in the DAAL application.

From Figure 9 we can see that for $N_v = 2^{30}$, our model distributed DAAL application is predicted to scale linearly. However, for $N_v = 2^{20}$, our model predicts that the application scaling at 8 compute nodes, and beyond 11 compute nodes the performance *decreases* with additional nodes. Although this is a rough estimate, such a graph can help the developer predict the performance of distributed DAAL applications (or any distributed process) on a large cluster.

Finally, the important thing to note is that the numbers presented here are *specific to our system and application*. The purpose of the exercise is the demonstrate how to estimate efficiency of an application on a large cluster environment based on the data for a small cluster.

Appendix B. Miscellaneous

B.1. MPI Pinning

In this section, we discuss the behavior of having multiple MPI processes running on the same Node. With MPI, you may put multiple processes on the same node using certain options for `mpirun` such as `-np` or `-ppn`. For more information on these options and others, refer to

In some MPI distributions, including Intel MPI, the MPI run-time "intelligently" allocates the resources (e.g. cores) to each process. For example, on a dual socket system that has two MPI processes running on it, cores are allocated (pinned) such that all threads in a single MPI process will stay within the same socket. In order to see how exactly the cores are being distributed to each MPI process, run the MPI application with the following Environment Variable.

```
user@localhost>  export I_MPI_DEBUG=4
user@localhost> mpirun -np 2 -host localhost ./my_MPI_app
...
[0] MPI startup(): Rank   Pid      Node name  Pin cpu
[0] MPI startup(): 0      6966     localhost  {0,1,2,3,4,5,6,7,16,17,18,19,20, ...
[0] MPI startup(): 1      6967     localhost  {8,9,10,11,12,13,14,15,24,25,26, ...
...
```

"Pin cpu" column list the processors associated with each rank. You can now cross reference the processor number in the above list with the processor enumeration scheme of your system. The command `cpuinfo` that prints this information calls the corresponding utility included in the Intel MPI library distribution. Because of this "intelligent" partitioning of systems, some applications and system architectures benefit from having multiple instances of the application running on the same system (for example, two MPI process running dual-socket systems may reduce cross-socket communication compared to just having one).

Intel DAAL applications respect the pinning set by MPI, so some distributed mode DAAL applications can benefit from multiple processes running on the same system.

B.2. Amdahl's Law

Amdahl's law states that the maximum performance scaling s of an application across s nodes is:

$$S(s) = \frac{1}{1-p+p/s}, \tag{2}$$

where `p` is the portion of an application that can be run in parallel across multiple nodes. To convert this equation to parallel efficiency `E(s)` We divide both sides by s:

$$E(s) = \frac{S(s)}{s} = \frac{1}{s(1-p)+p}. \tag{3}$$

For our correlation matrix application, parallel ratio `p` depends on two factors: number of vectors N_v and number of nodes s. Local computation and serialization can be done in parallel but communication and combination cannot, so

$$p = \frac{t_{\text{local}} + t_{\text{ser}}}{t_{\text{local}} + t_{\text{ser}} + t_{\text{comm}} + t_{\text{combine}}}. \tag{4}$$

Next, we must find how each of the times varies with two parameters of interest: N_v. Note that t_{local} scales linearly with N_v as we can observe from Figure 7. For t_{comm} and t_{combine}, doubling s would result in double the number of partial results, which will double t_{comm} and t_{combine}. (Note: since this is an approximation, we are ignoring the effect of the partial result from rank 0, which does not require deserialization). Thus, we would expect a linear relationship between t_{comm} and t_{combine} and s. Therefore:

$$t_{\text{local}} = c_1 N_v, \tag{5}$$

$$(t_{\text{local}} + t_{\text{combine}}) = c_2 s, \tag{6}$$

$$p(N_v, s) = \frac{c_1 N_v + t_{\text{ser}}}{c_1 N_v + t_{\text{ser}} + c_2 s}. \tag{7}$$

Here, `c_1` and `c_2` are constants. Using the data from Figure 7, we can solve for `c_1` and `c_2`. Note that for different systems and algorithms, the numbers will be different. By substituting 7 into the equation (3) we can approximate the expected parallel efficiency as a function of N_v for any s.