Searching for the Optimal Data Partitioning Shape for Parallel Matrix Matrix Multiplication on 3 Heterogenous Processors

Ashley DeFlumere, Alexey Lastovetsky
School of Computer Science and Informatics
University College Dublin
Belfield, Dublin 4, Ireland
ashley.deflumere@ucdconnect.ie, alexey.lastovetsky@ucd.ie

***Abstract*—Parallel Matrix-Matrix Multiplication (MMM) is a fundamental part of the linear algebra libraries used by scientific applications on high performance computers. As heterogeneous systems have emerged as high performance computing platforms, the traditional homogeneous algorithms have been adapted to these heterogeneous environments. Although heterogeneous systems have been in use for some time, it remains an open problem of how to optimally partition data on heterogeneous processors to minimize computation, communication, and execution time. While the question of how to subdivide these MMM problems among heterogeneous processors has been studied, the underlying assumption of this prior study is that the data partition shape, the layout of the data within the matrix assigned to each processor, should be rectangular, i.e. that each processor should be assigned a rectangular portion of the matrix to compute.**

Our previous work in this area questioned the optimality of this traditional rectangular shape and studied this partition shape problem for two processors. In that work, we proposed a novel mathematical method for transforming partition shapes to decrease communication cost and an analytical technique for determining the optimal shape.

In this work, we extend this technique to apply to three and more heterogeneous processors. While applying this method to two processors is relatively straightforward, the complexity grows immensely when considering three processors. With this complexity in mind, we propose a hybrid of experimental and analytical techniques. We postulate that a small number of partition shapes are potentially optimal, and perform extensive testing using a computer aided method to apply our previously developed analytical technique, without finding a counterexample. We identified six data partition shapes which are candidates to be the optimal three processor shape.

***Index Terms*—Parallel Matrix Multiplication; Matrix Partitioning; Heterogeneous Computing; High Performance Computing**

I. Introduction

Partitioning Parallel Matrix-Matrix Multiplication (MMM) over an arbitrary numbers of processors has been the subject of extensive study. For homogeneous systems, when the number of processors is a perfect square, the optimal solution is straight forward and well known. Considering arbitrary numbers of homogenous processors, however, is more challenging. For heterogeneous systems, the problem is even more substantive. First, heterogeneity comes in various forms. A collection of processors can be heterogeneous in their processing speeds, their communication interconnect or some combination of both [1]. In this paper, we will deal with heterogeneity in the processing speeds of the various processors.

Heterogeneous systems have become popular as high performance scientific computing platforms, and as such have been the subject of much research. Finding the optimal data partitioning for parallel MMM on these machines remains an open problem. The bulk of the previous study has focused on finding the optimal rectangular data partition, *i.e.* a partition in which each processor has a rectangular portion of the matrix to compute [2] [3] [4] [5] [6]. This is a difficult question, and indeed, finding the optimal rectangular partition for an arbitrary number of heterogeneous processors has been shown to be an NP-complete problem [7]. While many solutions have been proposed to efficiently find the best rectangular solution to the partitioning problem, the assumption that the solution should be rectangular has not been studied.

In our previous work [8], we questioned the optimality of the traditional rectangular shape. We began our study with the simple base case of two processors. Studying the case of a small number of heterogeneous processors is immediately relevant to several types of systems. In addition to being useful in its own right, we may view each "processor" as an abstract concept of a compute node or a group of tightly coupled computing devices. In [9], the authors use the concept of abstract logical processors to model GPU-CPU hybrid systems, where each logical processor represents an independent group of tightly coupled devices such as cores on the same socket or a GPU and its host core. In this approach, a modern hybrid compute node will be modeled by a small number of abstract heterogeneous processors. Additionally, such abstract processors may model several clusters, of differing computational speed, being used in concert.

We proved that for two processors, in some cases of processor speed ratios and MMM algorithms, a non-rectangular shape is actually optimal. Specifically, the Square-Corner partition, comprised of a slower processor assigned a square of data in some corner of the matrix, and a faster processor computing the non-rectangular remainder, was optimal when

the processing speed ratio was greater than three to one.

In order to demonstrate this we developed a novel mathematical technique called the Push operation. This technique allows us to begin with any random partition shape and transform its shape, while guaranteeing to never increase the communication cost or the execution time. When the partition shape can no longer be improved, it is a candidate to be the optimal shape.

While previous study [10] [11] on the Square-Corner partition had shown it superior to the traditional rectangular approach, [8] further proved that it was superior to any other possible shape, i.e. that it is the global optimum.

In this paper, we look to expand this work to cover three and more processors. The ultimate aim is not to show that a particular novel shape is better than the traditional shape, but to find the optimal shape. However, the complexity of the three processor case is far greater than for two processors. Ensuring, in the two processor case, that the Push operation forms one of the candidate shapes from any possible arbitrary starting partition shape was obvious upon inspection, and easily proven mathematically. For three processors, it is not immediately obvious that some arbitrary partition that cannot be reduced using Push does not exist. For this reason, we suggest a hybrid approach between our analytical technique and a newly developed experimental technique. We extensively test many random starting partitions for a representative sample of processor ratios, without finding a counterexample to our postulate that the three processor Push always reduces to some recognizable shape.

This new experimental technique is one of the contributions of this paper. The codebase begins with a random, arbitrary partition shape, and repeatedly applies the Push operation, in a random order and direction. To find the optimal shapes, we must only consider the partition shapes output by the experimental technique, as we know no other shape can be better in terms of communication and execution time. A fundamental requirement of this program is that it must also be applicable beyond the three processor case. It can easily be adapted to form partition shapes for any number of processors.

Using this new approach, we were able to identify four general partition shapes, or archetypes, for the case of three heterogeneous processors. These archetypes each represent several different partition shapes which fall under their general description. We prove that three of the archetypes can all be further improved to become data partition shapes encompassed by the fourth archetype.

Finally, we enumerate the data partition shapes which fall under the optimal archetype and find the canonical form for each, reducing the search for the optimal shape from millions of arbitrary arrangements of elements to just six well defined shapes.

II. Related Work

Our previous work in this area was a detailed study of data partitioning shapes for parallel MMM with two heterogeneous processors. We searched for the optimal data partitioning shape under five parallel MMM algorithms, and for all ratios between processing speeds.

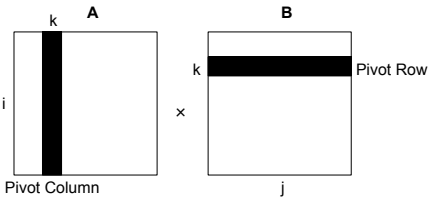

Fig. 1. Pivot row and column, k, of the kij algorithm. Every element of C is updated before k is moved to the next row and column.

To model the processor communication we used the linear Hockney Model [12], $T_{comm} = \alpha + \beta \times M$.

The kij algorithm is well-known and used by such software as ScaLAPACK [13] to compute MMM. It is the manner of computation for all algorithms presented in this paper. The kij algorithm for MMM is a variant of the triply nested loop algorithms. The three for loops are nested and iterate over the line $C[i,j] = A[i,k] * B[k,j] + C[i,j]$. The k variable represents a "pivot" row and column as shown in Fig. 1. For each iteration of k, every element of the result Matrix C is updated, incrementally obtaining the final value.

If the processor assigned to calculate an element of Matrix C has not been assigned some element in the corresponding pivot column or row, that element must be communicated to the processor.

The five MMM algorithms considered were:

1) *Serial Communication with Barrier (SCB)* All data is sent by each processor *serially*, and only once communication completes does the computation proceed in parallel on each processor.
2) *Parallel Communication with Barrier (PCB)* All data is sent among processors in *parallel*, and only once communication completes does the computation processed in parallel on each processor.
3) *Serial Communication with Bulk Overlap (SCO)* All data is sent by each processor *serially*, while in *parallel* any elements that can be computed without communication are computed. Only once both communication and overlapped computation are complete does the remainder of the computation begin.
4) *Parallel Communication with Bulk Overlap (PCO)* All data is sent among processors in *parallel*, while in *parallel* any elements that can be computed without communication are computed. Only once both communication and overlapped computation are complete does the remainder of the computation begin.
5) *Parallel Interleaving Overlap (PIO)* At each step data is sent a row and a column (or k rows and columns) at a time by the relevant processor(s) to all processor(s) needing that information, while, in parallel, all processors compute using the data sent in the previous step.

For each of these five MMM algorithms we created an analytical model of total execution time on two heterogeneous

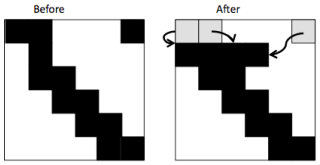

Fig. 2. A two processor partition shape is transformed using the Push Down operation.

processors as a function of processing speed, communication volume, communication bandwidth, the number of computations per matrix element, and the ratio between computation and communication speed [14]. For each algorithm, using these models, we demonstrated that the total execution time decreases, or at worst remains unchanged, when the total volume of communication is decreased and the computation time is unchanged.

To determine the optimal data partitioning shape, we created a mathematical method called the Push. This method takes an input partition shape, and provides an output partition shape which is guaranteed to have the same, or lower, volume of communication. We took the three general partition shapes output by this method, and analyzed them using the five parallel algorithms for all processing speed ratios. Of these three general shapes, two, the Straight-Line and the Square-Corner, were shown to always be superior to the third, the Rectangle-Corner. Additionally, the non-traditional, non-rectangular Square-Corner was found to be the optimal partition shape for:

- all processor power ratios when bulk overlapping communication and computation (SCO, PCO)
- processor power ratios greater than $3:1$ when placing a barrier between them or using interleaving overlap (SCB, PCB, PIO)

A *Push* operation acts on a single row or column of the matrix, k, to remove elements belonging to a single *active* processor, X. The row or column k must be the edge of Processor X's enclosing rectangle. An *enclosing rectangle* is an imaginary rectangle drawn around the elements assigned to a given processor, which is strictly large enough to encompass all such elements. The elements of X are moved from k into other rows and columns in the direction specified by the Push, *i.e.* Up, Down, Left, or Right. For example, in a Push Down ($\downarrow$) the elements of X are moved from k, into the rows below k. This is illustrated further in Fig. 2. Rules exist which limit what elements may be moved to which rows and columns, in order to guarantee that the Push operation will never increase the volume of communication or execution time.

III. Problem Statement

The problem of finding the optimal data partitioning shape for three heterogeneous processors is significantly more complex than with two processors. While expanding the rules of the Push technique to be applicable to N processors, we must consider not only where the elements of the active processor from row or column k may go, but whether the elements of other processors may be moved to row or column k. This added constraint makes it difficult to prove mathematically that some arbitrary arrangement of elements that cannot be Pushed does not exist.

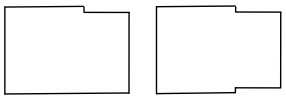

Fig. 3. An asymptotically rectangular partition shape, left, and a shape that is not asymptotically rectangular, right. Only a single row or column may be less than the length of the edge of the enclosing rectangle.

The aim of this paper is to apply the Push technique to three processors, with the help of a computer-aided method, to find a set of partition shapes which should be further analyzed, while discarding those shapes and non-shapes (arbitrary arrangements of elements) that do not need to be considered. Those partition shapes which warrant further study, known as the candidate partition shapes, are equal or better than all other shapes and non-shapes in terms of execution time.

IV. Theoretical Results

Throughout, we will make several assumptions, as follows:

1) Matrices A, B and C are square, of size $N \times N$, and identically partitioned among Processors P, R, and S, represented in figures as white, gray and black, respectively.
2) Processor P computes faster than Processors R and S by ratio, $P_r : R_r : S_r$, where $S_r = 1$.
3) All Processors may communicate with all other Processors, with no constraints on network topology.
4) Partition shapes are referred to as *rectangular* if they are asymptotically rectangular, *i.e.* if it contains at most a single side which has a single row or column that is less than the length of the that side of the shape, see Fig. 3.

Formally, each element of an $N \times N$ matrix is of the form (i, j) and a data partition shape is a function, such that,

$$q(i,j) = \begin{cases} 0 & \text{if } (i,j) \in R \\ 1 & \text{if } (i,j) \in S \\ 2 & \text{if } (i,j) \in P \end{cases}$$

In this section, we formally define the three processor Push and update the performance models for each of the five MMM algorithms for three processors.

A. Three Processor Push Definition

A Push is an atomic operation performed on a partition shape, q, and producing a new partition, q_1. A Push may *not* enlarge the enclosing rectangle of any processor.

Here, we formally describe the $\downarrow$ direction of the Push operation for three processors. The $\uparrow, \leftarrow$ and $\rightarrow$ directions are similar.

Consider three processors, P, R and S. Processor P is equal or faster in processing speed than each processor, R

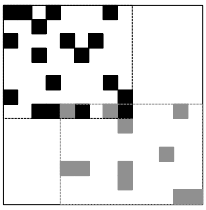

Fig. 4. A matrix partitioned among 3 processors, white, gray and black. The dotted lines are the enclosing rectangles around the black and gray processors. The enclosing rectangle for the white processor is the entire matrix.

and S. Processor R is the *active* processor. Each processor has an enclosing rectangle, r, s and p respectively. The row and column values of the dimensions of the enclosing rectangle of Processor X are known, in clockwise order, as $x_{top}, x_{right}, x_{bottom}$ and x_{left}.

The Push$\downarrow$ operation creates a new partition shape, q_1 from the existing one, q, by *cleaning* the top row of the enclosing rectangle, r_{top}, of all elements assigned to Processor R. Processor R is assigned a corresponding number of elements in the rows below. The Processor, either S or P, which owned those elements newly assigned to R, are assigned the elements in r_{top}.

When Pushing three processors, there are six possibilities for a legal Push, *i.e.* a Push which decreases, or at least does not increase, the volume of communication. The volume of communication of any data partition shape q is given by

$$\text{VoC} = \sum_{i=1}^{N} N(c_i - 1) + \sum_{j=1}^{N} N(c_j - 1) \tag{1}$$

$c_i - \#$ of processors assigned elements in row i of q
$c_j - \#$ of processors assigned elements in column j of q

1) Type One - Decreases VoC: For each element assigned to Processor R in r_{top}, Processor R is assigned an element in the below rows and columns already containing elements of Processor R.

For each element which has been reassigned to R, the Processor previously assigned that element is given some unassigned element (r_{top}, j). Prior to the Push, this Processor must have already had an element in row r_{top} and in column j.

2) Type Two - Decreases VoC: For each element assigned to Processor R in r_{top}, Processor R is assigned an element in the rows below. Elements may go to some number, l, of rows and columns which did not already contain elements of Processor R, *dirty*ing those rows and columns, if l or more rows and columns are also *clean*ed of R.

For each element which has been reassigned to R, the Processor previously assigned that element is given some unassigned element (r_{top}, j). Prior to the Push, this processor must already have had an element in row r_{top} and in column j.

3) Type Three - Decreases VoC: For each element assigned to Processor R in r_{top}, Processor R is assigned an element in the rows and columns below that already contain elements of Processor R.

For each element which has been reassigned to R, the Processor previously assigned that element is given some unassigned element (r_{top}, j). Prior to the Push, it is not necessary for this Processor have had an element in r_{top} or j, provided the number of rows and columns dirtied, l, is less than the number of rows and columns cleaned.

4) Type Four - Decreases VoC: For each element assigned to Processor R in r_{top}, Processor R is assigned an element in the rows below. Elements may go to some number of rows and columns, l, which did not already contain elements of Processor R, dirtying those rows and columns, if l or more rows and columns are also cleaned of R.

For each element which has been reassigned to R, the Processor previously assigned that element is given some unassigned element (r_{top}, j). Prior to the Push, it is not necessary that this Processor have already had an element in r_{top} or j, provided the number of rows and columns dirtied, l, is less than the number of rows and columns cleaned.

5) Type Five - Unchanged VoC: For each element assigned to Processor R in r_{top}, Processor R is assigned an element in the rows below. A single row or column not containing elements of Processor R may be dirtied.

For each element which has been reassigned to R, the Processor previously assigned that element is given some unassigned element (r_{top}, j). Prior to the Push, this Processor must have been assigned an element in row r_{top} and in column j.

6) Type Six- Unchanged/Decrease VoC: For each element assigned to Processor R in r_{top}, Processor R is assigned an element in the rows below. A single row or column not containing elements of Processor R may be dirtied.

For each element which has been reassigned to R, the Processor previously assigned that element is given some unassigned element (r_{top}, j). Prior to the Push, it is not necessary that this Processor have had an element in r_{top} or j, provided the number of rows and columns dirtied, l, is less than or equal to the number of rows and columns cleaned.

B. Parallel MMM Algorithm Performance Models

These performance models define execution time, for each MMM algorithm, as a function of communication and computation time. This section asserts that if communication time is decreased, for each of the algorithms modeled, the execution time will either decrease or remain unchanged.

1) Serial Communication with Barrier:

$$T_{exe} = T_{comm} + T_{comp} \tag{2}$$

$$T_{comm} = \Big(\sum_{i=1}^{N} N(p_i - 1) + \sum_{j=1}^{N} N(p_j - 1) \Big) \times T_{send} \tag{3}$$

where,

$p_i = \#$ of processors assigned elements in row i
$p_j = \#$ of processors assigned elements in column j
$T_{send} = \#$ of seconds to send one element of data

As Push is designed to clean a row or column of a given processor, it will decrease p_i and p_j, lowering communication time, and thereby execution time. At worst, it will leave p_i and p_j unchanged.

2) Parallel Communication with Barrier:

$$T_{exe} = T_{comm} + T_{comp} \quad (4)$$
$$T_{comm} = \max(d_P, d_R, d_S) \quad (5)$$

Here, d_X is the time taken, in seconds, to send all data by Processor X and is formally defined below.

$$d_X = \big((N \times i_X + N \times j_X) - \in X\big) \times T_{send} \quad (6)$$

where,

$i_X = \#$ of rows containing elements of Processor X
$j_X = \#$ of columns containing elements of Processor X
$\in X = \#$ of elements assigned to Processor X

Each Push operation is guaranteed, by definition, to decrease or leave unchanged, either i_X or j_X, or both.

3) Serial Communication with Bulk Overlap:

$$T_{exe} = \max\big(d_P + d_R + d_S, \max(o_P, o_R, o_S)\big) + \max(c_P, c_R, c_S) \quad (7)$$

where,

$o_X = \#$ of seconds to compute the overlapped computation on Processor X
$c_X = \#$ of seconds to compute the remainder of the data on Processor X

Each Push operation is guaranteed, by definition, to decrease or leave unchanged, i_X and j_X of d_X for the active processor. It also, by definition, will not increase d_X for either inactive processor.

4) Parallel Communication with Overlap:

$$T_{exe} = \max\big(\max(d_X, d_R, d_S), \max(o_P, o_R, o_S)\big) + \max(c_P, c_R, c_S) \quad (8)$$

Each Push operation is guaranteed, by definition, to decrease or leave unchanged, i_X and j_X of d_X for the active processor. It also, by definition, will not increase d_X for either inactive processor.

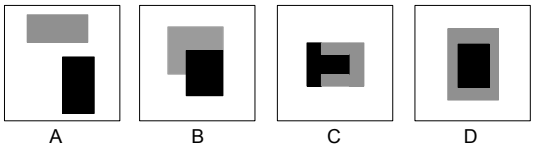

Fig. 5. The 4 possible general partition shape archetypes found using the Push program. Each of these archetypes has multiple shapes which must be explored for each ratio. Although all shape archetypes are pictured as non-rectangular, Archetype A includes all the traditional rectangular partitions in it's general description.

5) Parallel Interleaving Overlap:

$$T_{exe} = \text{Send } k + \sum_{i,j,k=1}^{N} \max\left(\Big(N \times \big((p_i - 1) + (p_j - 1)\big)\Big) \times T_{send}, \max(k_P, k_R, k_S)\right) + \text{Compute}(k+1) \quad (9)$$

where,

$p_i = \#$ of processors assigned elements in row i
$p_j = \#$ of processors assigned elements in column j
$k_X = \#$ of seconds to compute step k on Processor X

V. Using Push to find Optimal Shapes

The Push operation is used to form three processor data partition shapes for MMM which minimize communication cost and execution time. The aim of this work is to identify data partition shapes which are better, or at least no worse, than any other shape or non-shape (arbitrary arrangement of elements). These are labeled as *candidates* to be the optimal shape, under specific conditions. However, it is difficult to show mathematically that the Push operation will always give some recognizable shape when used on data partitions with three or more heterogeneous processors. We introduce in this section an experimental technique to accompany our analytical methods. This allows us to assert that no non-shape, an arbitrary arrangement of elements with no defined borders or single contiguous allocation of elements, will be superior to the condensed shapes we present.

Postulate 1 (Three Processor Push): There exists no arrangement of elements among three heterogeneous processors in an $N \times N$ matrix which cannot be improved with the Push operation, except those arrangements of shapes defined in Fig. 5.

To motivate Postulate 1, we present the problem as a Deterministic Finite Automaton. A DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

1) Q is the finite set of states, the possible data partitioning shapes
2) Σ is the finite set of the alphabet, the processors and the directions they can be Pushed

3) δ is $Q \times \Sigma \rightarrow Q$ the transition function, the Push Operation
4) q_0, the start state, chosen at random
5) F is $F \subseteq Q$, the accept states, candidate partitions to be the optimum

The finite set of states, Q, is every possible permutation of the three processor's elements within the $N \times N$ matrix. Therefore the number of states in the DFA is dependent on the size of the matrix, the number of processors and the relative processing speeds of those processors.

The finite set, Σ, called the alphabet, is the information processed by the transition function in order to move between states. Legal input symbols are the active processor being Pushed, X, and the direction the elements of Processor X are to be moved, *i.e.* Up, Down, Right or Left.

The transition function δ is the Push operation. This function processes the input language Σ and moves the DFA from one state to the next, and therefore the matrix from one partition shape to the next. The implementation of the transition function is discussed further in the next section.

Finally, the accept states F are those fixed points in which no Processor X may be Pushed in any direction. These states, and their corresponding partition shapes, must be studied further.

VI. Program Construction

As the primary contribution of this paper, we designed a program which implements this DFA. It begins with a random q_0 start state. It moves from between states by following the Push operation transition function. When no transitions remain, the final state is a candidate for the optimal partition shape. This program is designed to be executed many thousands of times with various inputs. In this way, we may convince ourselves that no partition exists that is better than the final partition shapes we have discovered.

A. Program Inputs

The program takes as input the number of processors, 3, their relative processing speeds, $P_r : R_r : 1$, and the size of the matrix, N. One of the slower processors, R or S, is randomly selected to be the active processor first for the Push operations.

1) Randomizing Push Direction: Part of the difficulty of searching for potentially optimal data partition shapes is ensuring all possible shapes are considered. Preconceived notions about what is likely to be optimal should not determine how the program searches for these potentially optimal shapes. For each new starting state, our program selects a random number of directions (1, 2, 3 or 4) to Push the active processor. The Push directions are then randomly selected. For example, if 2 is selected as the number of directions, then Up and Left, or Down and Left, and so forth, might be selected as Push directions. Finally, the order of Push operations is randomly selected. In this way, we cover such disparate cases as one Push direction only, two Push directions in which one direction is exhausted before the other begins, or four Push directions where each Push direction is interleaved with the others.

2) Randomizing q_0: At the beginning of the program, all elements are assigned to the fastest processor, P. Each subsequent processor, X, is considered in turn. Random integer values are selected for row and column, separately, and if the element at (i, j) has not already been assigned to a processor other than Processor P, it is assigned to the Processor X.

B. Program Operation

In this section, we describe the way in which the program searches for a legal Push on the active Processor, X, in a given direction.

We define several metrics of a partition shape:

For row i,

$$row(q, i, X) = \begin{cases} 0 & \text{if } (i, \cdot) \text{ of } q \textit{ no} \text{ elements of } X \\ 1 & \text{if } (i, \cdot) \text{ of } q \text{ has elements of } X \end{cases}$$

For column j,

$$col(q, j, X) = \begin{cases} 0 & \text{if } (\cdot, j) \text{ of } q \textit{ no} \text{ elements of } X \\ 1 & \text{if } (\cdot, j) \text{ of } q \text{ has elements of } X \end{cases}$$

As an example, we discuss a Push Down on active Processor R, but the other directions are similar.

Formally, $\downarrow q(R) = q_1$ where,

Initialize $q_1 \leftarrow q$
$(g, h) \leftarrow (r_{top} + 1, r_{left})$
for $j = r_{left} \rightarrow r_{right}$ **do**
  if $q(r_{top}, j) = 0$ **then**
    {Element is dirty, clean it}
    $(g, h) \leftarrow$ find (g, h) {Function defined below}
    if $q(g, h) = 1$ **then**
      $q_1(r_{top}, j) \leftarrow 1$ {Cleaned element assigned to S}
    end if
    if $q(g, h) = 2$ **then**
      $q_1(r_{top}, j) \leftarrow 2$ {Cleaned element assigned to P}
    end if
    $q_1(g, h) \leftarrow 0$ {Put displaced element in new spot}
  end if
  $j \leftarrow j + 1$
end for

The function find(g,h) searches for a suitable swap of elements according to defined Push Types. This is the algorithm for finding a Type One Push, the other Types are similar.

findTypeOne(g, h) {Look for a suitable slot to put element}
for $g \rightarrow r_{bottom}$ **do**
  for $h \rightarrow r_{right}$ **do**
    if $q_1(g, h) \neq 0$ && $(row(q, r_{top}, (q_1(g, h)) = 1 \parallel col(q, j, (q_1(g, h)) = 1)$ && $(row(q, g, R) = 1 \parallel col(q, h, R) = 1$ **then**
      return (g, h)
    end if
    $h \leftarrow h + 1$
  end for
  $h \leftarrow k_{left}$
  $g \leftarrow g + 1$
end for
return $q_1 = q$ {No Type One Push$\downarrow q(R)$ possible}

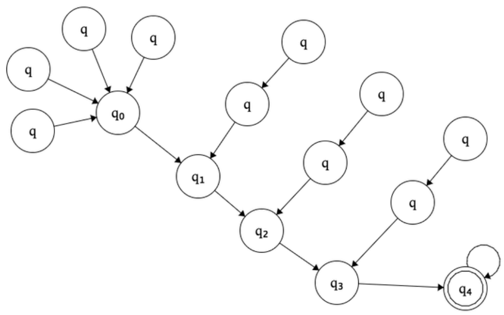

Fig. 6. Those states, q, which have transition arrows to states, q_0 through q_4, which are a known (experimentally found) path to an accept state. If it is possible to transition to any state on this path from q, then we denote q to be considered experimentally, and not a possible counterexample.

The full code base is available for inspection at http://www.hcl.ucd.ie

C. End Conditions

In order to implement the Push, we must strictly define the conditions under which a partition is considered fully Pushed. In the theoretical Push, a partition is considered fully Pushed, or condensed, when the elements of no processors, except the largest processor, may be legally moved in any Push direction.

The implementation of our program first determines the valid directions of Push for a given processor in a given run. A partition fully condensed if there are no available Push operations in any direction in that set of possible directions.

VII. Experimental Results for 3 Processors

The size of the test matrix chosen for the experiments must be large enough to possess the granularity of elements required to form a variety of shapes, and be considered representative of any value of N. However, the larger the matrix size N, the larger the set of possible states, Q, and therefore more experimental runs are necessary to appropriately cover them all. To balance these two requirements, we chose $N = 1000$.

The processor ratios chosen for study were 2:1:1, 3:1:1, 4:1:1, 5:1:1, 10:1:1, 2:2:1, 3:2:1, 4:2:1, 5:2:1, 5:3:1, 5:4:1. For each ratio, the DFA implementation was run approximately 10,000 times. As the DFA is not a simulation of actual MMM on parallel processors, but searching for partition shapes which cannot be improved using the Push operation, it is run on a single processor. Multiple instances of the program were run on multiple processors to increase the speed at which data was collected. The DFA was run on a small cluster of Dell Poweredge 750 machines with 3.4 Xeon processors, 256 MB to 1 GB of RAM, and 1 MB of L2 cache.

A. Experimental Thoroughness

The problem space, the number of possible arrangements of elements, given an N size matrix, and 3 processors is 3^{N^2}. However, we can further restrict this space to ensure the number of states, Q, covered in our experiments is attainable. First, we assert that because we have a set number of elements

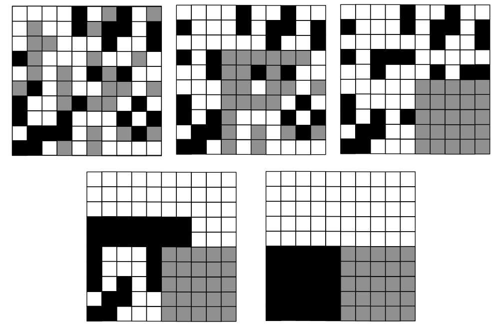

Fig. 7. An example run of the DFA at approximate steps 1, 500, 1000, 1500 and 2100.

assigned to each processor, $\in P, \in R$ and $\in S$, we can say the total number of possible states Q is $\frac{N^2!}{(\in P!)\times(\in R!)\times(\in S!)}$.

A state, q, has been considered if it is a q_0 for an experiment, any state, q_x, passed through during an experiment, or any state with a transition arrow leading to either q_0 or q_x. This is shown in Fig. 6. This significantly reduces the number of states that must be considered by the Push program code.

B. Example Run

This example run is of ratio 2:1:1, $N = 1000$, with Processor R randomly selected to be moved in two directions, down and right, and Processor S randomly selected to be moved in two directions, down and left. Fig. 7 shows the approximate partition shape at approximate steps 1, 500, 1000, 1500, and 2100. These figures are shown at $\frac{1}{100}^{th}$ granularity, *i.e.* each box shown is a 100×100 square of individual elements. The colors given represent the Processor assigned the majority of the elements in that 100×100 square.

C. Results

The next challenge was to determine if the final states, F, were condensed shapes and whether a counterexample had been found. To make the output manageable, we grouped every state, q, in F, into a shape archetype. A *shape archetype* is a general description of q, based on the layout and overlap of the enclosing rectangles of R and S, and the number of *corners* in R and S. The number of corners is equal to the number of interior angles of a processor's shape. The minimum number of corners for any of the output shapes was four. Further definition of corners can be found in Section VIII-A.

For all experimental processor ratios, the DFA program revealed four general partition shape archetypes. These partition archetypes are shown in Fig. 5.

D. Archetype A - No Overlap, Minimum Corners

In Archetype A partitions, the enclosing rectangles of Processors R and S do *not* overlap. Processors R and S are each rectangular, possessing the minimum number of corners (four). Processor P is assigned the remainder of the matrix. Depending on the dimension and location of Processors R and S, the matrix remainder assigned to Processor P may be either rectangular or non-rectangular.

If Processor P is rectangular, the entire partition shape, q, is rectangular. Otherwise, q is a non-traditional, non-rectangular shape. It is important to note that although these two partition shapes seem disparate at first glance, they are similar in their description of enclosing rectangles and corners, and so are grouped together.

E. Archetype B - Overlap, L Shape

In Archetype B partitions, the enclosing rectangles of Processors R and S partially overlap. One processor, shown in Fig.5 as Processor S, is rectangular, having four corners. Processor R has six corners, and is arranged in an "L" shape adjacent to the rectangle shape of Processor S. Processor P is assigned the remainder of the matrix.

F. Archetype C - Overlap, Interlock

In Archetype C partitions, the enclosing rectangles of Processors R and S partially overlap. Neither processor has a rectangular shape. Each processor has a minimum of six corners. Processor P is assigned the remainder of the matrix, which may be rectangular or non-rectangular.

We note that in all experimentally found examples of Archetype C, if the shapes of Processors R and S were viewed as one processor, they would be rectangular.

G. Archetype D - Overlap, Surround

In Archetype D partitions, the enclosing rectangle of one processor, shown in Fig. 5 as Processor S, is entirely overlapped, or surrounded, by Processor R's enclosing rectangle. Processor S has four corners, while Processor R has eight corners. Processor P is assigned the remainder of the matrix, which may be rectangular or non-rectangular.

VIII. Analysis of Shape Archetypes

In this section, we demonstrate that partition shapes of Archetypes B, C and D may all be transformed, without worsening their volume of communication or execution time, into Archetype A partitions. Then we may eliminate those actual partition shapes which fall under those Archetypes, and consider only those shapes which are of Archetype A.

Theorem 8.1: In a partition among three heterogeneous processors, the position of the two smaller processor shapes, within the context of the larger matrix, does not affect the total volume of communication, if the position of the two smaller shapes do not change relative to each other.

Proof: Consider the shapes of Processor R and S to be one continuous shape. Their position relative to each other will not change, so moving this combined shape is analogous to moving a single small processor in a two processor data partition. This is known not to increase the volume of communication [8]. ■

A. Taxonomy of Corners

A corner is a point in a partition shape, q, of a single processor at which the previously constant coordinate, x or y, of the edge changes, and the other coordinate, either x or y becomes a constant.

Each shape has four edges to consider, even if parts of each edge lie on different rows or columns. An edge is any row or column within a partition shape that does not have another row or column (depending on whether a vertical or horizontal edge) containing it's processor's elements between it and the side of the matrix it is named after. The sides of the matrix, beginning at the bottom and moving clockwise are named x, y, z, and w.

Each processor, P, R and S, has a minimum of four corners. Each edge is denoted using the notation P_{x1}, P_{x2}, P_{y1}, P_{y2} and so on. This is shown in Fig. 8. When a shape has the minimum number of corners, then each corner may be referred to by two notations, *i.e.* $P_{y1} = P_{z1}, P_{z2} = P_{w1}, P_{w2} = P_{x2}$ and $P_{x1} = P_{y2}$. Note that for vertical edges, y and w, points are given top to bottom, and for horizontal edges, x and z, are given left to right.

If, for a given processor, the edge points are not equal to their corresponding adjacent edge points, then at least one extra corner must exist along those two edges.

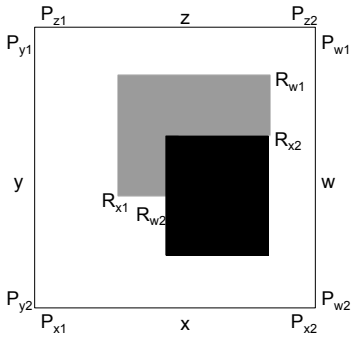

Fig. 8. An Archetype B partition shape shown with the corner notation. Not all points are labeled, but all points follow the pattern shown by points labeled for Processor P.

B. Archetype B to Archetype A

Theorem 8.2: Any Archetype B partition shape, q, may be transformed into a Archetype A partition shape, q_1, without increasing the volume of communication of the shape

Proof: The Archetype B partitions have two important cases to consider. First, those where the combined width or height of the smaller processors is equal to N and secondly, those in which the combined length is less than N. These are shown in Fig. 9.

For both cases of Archetype B partitions, we will apply a Push-like transformation to Processor R, the "L" shape, along one of the planes with the extra corner. In Fig. 9 this is either the x or w sides, so the elements of Processor R may be moved in either the Left ($\leftarrow$) or Up ($\uparrow$) directions. This is not strictly a Push operation, as the enclosing rectangle for R will

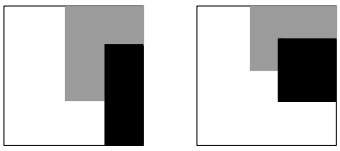

Fig. 9. The two possible cases of an Archetype B partition shape. In the first case, left, the combined length of the two shapes is N, the full length of the matrix. In the second case, right, the combined length of the two shapes is less than N.

be expanded in one direction. Because the enclosing rectangle is also being diminished in another direction, we can show that the volume of communication has not increased.

In the first case, to move the elements of R, only one transformation direction is available because the length N of the combined rectangles does not allow room for additional swaps. In the example of Fig. 9 we choose the Left ($\leftarrow$) direction.

For each column transformed to remove elements of R, at most one column previously not containing R will have elements of R introduced. This is assured, by definition, by virtue of the existence of the corner:

$$R_{w1} \rightarrow R_{x2} < R_{y1} \rightarrow R_{y2} \tag{10}$$

where,

$$R_{w1} \rightarrow R_{x2} = \# \text{ of rows separating } R_{w1} \text{ and } R_{x2}$$
$$R_{y1} \rightarrow R_{y2} = \# \text{ of rows separating } R_{y1} \text{ and } R_{y2}$$

For the second case, the elements of R can be moved in either direction, as the combined length of both shapes is less than N, and therefore rows and columns exist in either direction into which elements of R can be moved. The direction of the Push-like transformation is decided by choosing that which requires the lower volume of elements to be moved. In example Fig. 9, we would first use Theorem 8.1 to move the entire shape of Processors R and S down in the matrix so that $S_{x2} = P_{x2}$, opening rows above Processor R so we may move elements in the Up ($\uparrow$) direction.

For each row transformed to remove elements of R, at most one row previously not containing R will have elements of R introduced. This is assured by definition, by virtue of the existence of the corner:

$$R_{x1} \rightarrow R_{w2} < R_{z1} \rightarrow R_{z2} \tag{11}$$

where,

$$R_{x1} \rightarrow R_{w2} = \# \text{ of rows separating } R_{x1} \text{ and } R_{w2}$$
$$R_{z1} \rightarrow R_{z2} = \# \text{ of rows separating } R_{z1} \text{ and } R_{z2}$$

For every row or column made dirty with R during these transformations, a row or column must have, by definition been made clean of R, so volume of communication is constant or decreasing. ■

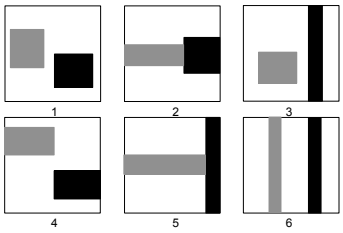

Fig. 10. The six candidate partition types found under Archetype A.

C. Archetype C to Archetype A

Theorem 8.3: Any Archetype C partition shape, q, may be transformed into a Archetype A partition shape, q_1, without increasing the volume of communication of the shape by applying the Push operation

Proof: By definition of this shape, valid Push operations remain, which if applied will result in an Archetype A partition. ■

Archetype C is the only archetype formed by our program on which there are valid Push operations remaining. These form as a result of the randomized Push direction algorithm, and is a necessary downside to truly considering every possible partition shape without preconceived notions of the final shape. Transforming partition shapes of this archetype is a simple matter of applying the Push operation in the direction not selected by the program.

In the program, this case is handled by a "beautify" function to return rectangular or asymptotically rectangular shapes, but is included here for comprehensiveness.

D. Archetype D to Archetype A

Theorem 8.4: Any Archetype D partition shape, q, may be transformed into an Archetype A partition shape, q_1, without increasing the volume of communication of the shape

Proof: In [8] it was proven that for two processors, the location of the smaller processor within the context of the larger matrix, does not effect the total volume of communication.

Consider the surrounding processor, in figures Processor R, and the inner processor, Processor S, to be a two processor partition in a matrix the size of Processor R's enclosing rectangle.

By [8] Theorem 3.4 Canonical Forms, move Processor S so that $R_{x2} = S_{x2}$.

We have created an Archetype B partition from an Archetype D, without increasing its volume of communication. By Theorem 8.2, it may be further reduced to Archetype A. ■

IX. Candidate Partitions

The candidate partition types included under Archetype A are seen in Fig. 10. In all six, Processors R and S are assigned rectangular portions of the matrix to compute. The these rectangles vary in length from, at the longest, N to, at the shortest, $\sqrt{\in X}$, *i.e.* the side of a square containing all the elements of X.

Each candidate shape pictured in Fig. 10 is representative of all shapes matching it's general description. The location within the matrix for each Processor R and S may be different than shown, and is a factor to consider when determining the optimal canonical partition shape. Here we formally define these 6 partition shape types, and describe which parts are fixed and which may be changed to create a valid partition shape of the same type. However, it is important to note that we do not assert that all valid partition shapes of the same type are necessarily equivalent. Indeed, in the next section we define the canonical, "best version", of each of the candidate types.

A. Formal Definition of Candidate Shapes

Listed here are the fixed points in the definition of each type. If a dimension or a relative location is left unspecified, that dimension may have any value from zero to the size of the matrix, N. A shape is considered rectangular if all processors are assigned a single rectangular portion of the matrix to compute. Shapes in which a single processor is assigned two or more rectangles to compute are non-rectangular.

Processors R and S will be referred to as having the dimensions R_{width}, R_{length} and S_{width}, S_{length} respectively. The value of R_{width} is derived from the distance between points R_{x1} and R_{x2} described in Section VIII-A. The value of R_{length} is the distance between points R_{y1} and R_{y2}. The values S_{width} and S_{length} are derived in the same manner.

A partition shape falls under the given type if it fulfills the listed criteria or can be rotated to meet the criteria. We normalize the dimension of the matrix, N, to be 1 for the following equations. For all types R and S are rectangular so $R_{width} \times R_{length} =\in R$ and $S_{width} \times S_{length} =\in S$.

Type One

$R_{width} + S_{width} < 1, R_{length} < 1, S_{length} < 1$

Type Two

$R_{width} + S_{width} = 1$

$R_{length} < 1, S_{length} < 1, R_{length} \neq S_{length}$

Type Three

$R_{width} + S_{width} < 1, R_{length} < 1, S_{length} = 1$

Type Four

$R_{width} + S_{width} = 1, R_{length} < 1$

$S_{length} < 1, R_{length} = S_{length}$

Type Five

$R_{width} + S_{width} = 1, R_{length} < 1, S_{length} = 1$

Type Six

$R_{width} + S_{width} < 1, R_{length} = 1, S_{length} = 1$

B. Finding the Optimal Version

Each of the candidate partition types has some leeway for difference either in the dimensions of rectangles R and S or in their location within the matrix. We define the canonical version for each candidate partition type. To minimize communication time, here we focus on minimizing the combined perimeters of rectangles R and S, under the constraints of each partition type. For the following proofs we normalize the size of the matrix to $N = 1$, and define

$$T = P_r + R_r + S_r \tag{12}$$

1) Splitting Type One: The Type One candidate partition type has two rectangles, each of which is less than the matrix width, 1, in both dimensions. We assert the minimum perimeter of a rectangle of fixed area occurs when width and height are equal, *i.e.* when the rectangle is a square. However, we notice that it may not always be possible to form two non-overlapping squares in an 1×1 matrix, even if we fix $R_{y1} = P_{y1}$ and $S_{x2} = P_{x2}$.

Theorem 9.1: The rectangles formed by Processors R and S may both be squares when $P_r > 2\sqrt{R_r}$.

Proof: The volume of elements assigned to each processor is equal to the Processors' ratio divided by the sum of the ratios, and multiplied by the total volume of elements in the matrix. The volume of elements assigned to each of the Processors, P, R and S, are $\frac{P_r}{T}$, $\frac{R_r}{T}$, and $\frac{1}{T}$, respectively. If we assume that both Processors R and S are squares, then the length of their sides will be $\sqrt{\frac{R_r}{T}}$ and $\sqrt{\frac{1}{T}}$ respectively. In order for the squares to fit in the 1×1 matrix without overlapping,

$$\begin{aligned} \sqrt{\frac{R_r}{T}} + \sqrt{\frac{S_r}{T}} &< 1 \\ R_r + 2\sqrt{R_r S_r} + S_r &< T \\ 2\sqrt{R_r} &< P_r \end{aligned}$$

■

For those ratios where $P_r < 2\sqrt{R_r}$, and two squares may not be formed, the optimal shape which still conforms to the Type One criteria must be found. We look to minimize the function,

$$f(x, y) = 2\Big(\frac{R_r}{Tx} + x + \frac{S_r}{Ty} + y\Big) \tag{13}$$

under the constraints,

$$\begin{aligned} 0 < \frac{R_r}{xT} &< 1 \\ 0 < \frac{S_r}{yT} &< 1 \\ x + y &< 1 \end{aligned}$$

The slope of the surface bounded by these constraints is increasing with x and y. Indeed the derivative of Equation 13 is positive, indicating it is increasing. To find the minimum of (13) then, we search along the lower bound when $x + y \approx 1$. This allows us to rewrite (13) as a function of x, set it's derivative equal to zero and solve for x. This gives $x = -\frac{\sqrt{R}-R}{R-1}$. Showing, for ratios such that $P_r < 2\sqrt{R_r}$, the optimal shape is two non-square rectangles R and S of combined width of approximately 1, but less than 1 by definition. The two optimal versions of Type One partitions can be seen in Fig. 11.

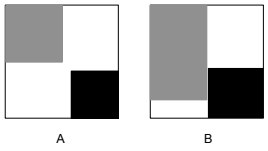

Fig. 11. On the left is a Type 1A partition shape, the Square-Corner, with Processors R and S each formed into a square. On the right is a Type 1B partition shape, the Rectangle-Corner, showing two non-square rectangles.

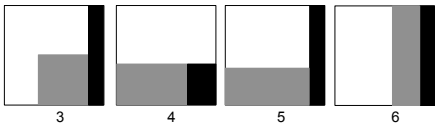

Fig. 12. The best versions of candidate partition types 3 to 6, *i.e.* each type in canonical form. Type 3 partition, the Square-Rectangle. Type 4 partition, the Block-Rectangle. Type 5 partition, the L-Rectangle. Type 6 partition, the Traditional-Rectangle.

2) Combining Type Two and Four: Partition shape Types Two and Four are similar, but Type Four is more rigid. In a Type Four partition, both dimensions of rectangles R and S are fixed, and only their relative location in the matrix may be altered. In Type Two, the total width of R and S is fixed, but the relative dimensions may change. A Type Two partition is improved, lowering the volume of communication, by transforming it into a Type Four partition by changing the relative widths so that $R_{height} = S_{height}$. The canonical form of the Type Four partition, the Block-Rectangle, is shown in Fig. 12, with $R_{y1} = P_{y2}$ and $S_{z1} = P_{z2}$.

3) Type Three Canonical Form: The Type Three partition has a rectangle of height N, and therefore of fixed width. The second rectangle is unfixed in both dimensions, and as shown above the optimal shape for a rectangle on length and width less than N is a square. It is possible to form a square and a rectangle for all ratios $P_r : R_r : 1$, without regard to which Processor, R or S, is the square. The canonical form of the Type Three partition, the L-Rectangle, is shown in Fig. 12, with $R_{x2} = S_{x1}$ and $S_{z1} = P_{z2}$.

4) Type Five and Type Six Canonical Form: In both Type Five and Type Six partitions, the height and width of both processors R and S is fixed, and only their relative position within the matrix may be changed. For Type Five, the L-Rectangle partition, we fix $R_{y1} = P_{y2}$ and $S_{z1} = P_{z2}$. In Type Six partitions, the Traditional-Rectangle we set $R_{x1} = P_{x2}$ and $S_{x1} = R_{x1}$, as seen in Fig. 12.

X. Analyzing Canonical Shapes to Find Optimal

Of the six potentially optimal partition shapes, at least one will be the optimum for a given set of factors. We must analyze these shapes to determine the optimal partition shape for all ratios $P_r : R_r : S_r$. This full analysis is beyond the scope of this paper, but the methodology is described here. As with the two processor case, for each MMM algorithm we refine the performance model to reflect the communication characteristics of the partition shape. The models are then compared algebraically and graphically to determine their relative cost in terms of execution time. For all ranges of ratios, the partition shape with the minimum execution time at a given value of ratios is said to be the optimum partition shape for those given values.

Other factors which determine the optimal shape are network topology and the ratio between communication and computation speed. In the case of three processors, there are two topologies to consider. First, we consider a fully connected topology, where each processor is able to send and receive data from all other processors. Secondly, there is the star topology, where one central processor may send and receive data with each of the other processors, but those two processors do not communicate with each other. The models presented earlier in this paper apply to the first, fully connected topology. It is obvious that the additional restriction of communication topology in the star will affect the which partition shape is the optimal.

A. Theoretical Comparison

Although the full analysis is beyond the scope of this paper, the algebraic and graphical comparison is shown here to motivate the use of these canonical shapes. We take as an example the SCB algorithm, Equation 2, the fully connected network topology, and the Square-Corner and Block-Rectangle partition shapes.

$$\text{Square Corner vs. Block Rectangle}$$
$$2N(R_{width} + S_{width}) < N(R_{length} + N)$$
$$\text{Normalize } N = 1$$
$$2(R_{width} + S_{width}) < R_{length} + 1$$
$$\sqrt{\frac{R_r}{T}} + \sqrt{\frac{1}{T}} < 1 - \frac{P_r}{T}$$

At this point it becomes clear that both the Square-Corner and the Block-Rectangle are dependent on R_r and P_r (due to the T variable). This can be seen in the graph of these two functions in Fig. 13.

B. Experimental Comparison

To further motivate and validate the contribution of this paper, we briefly present experimental results of the Square-Corner versus Block-Rectangle partition shapes for the SCB algorithm on a fully connected topology. Theoretically, we calculate that if it is possible to form a Square-Corner partition (due to processor speed ratios), for high heterogeneity ratios it will outperform the Block-Rectangular partition shape in terms of communication and execution time.

Experimental results were found using three identical processors. All ratios tested were such that $R_r = S_r$ for simplicity. Communication uses Open-MPI [15] and local matrix multiplications use ATLAS [16]. The processing speed ratio was controlled on the three nodes by using a CPU limiting program which monitors processes using the /proc filesystem. A process is allowed to run until a set fraction of CPU time

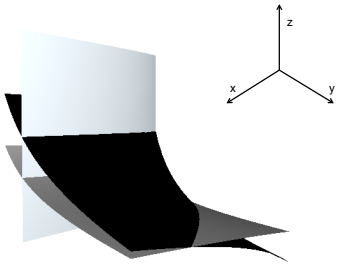

Fig. 13. The cost functions of the Square-Corner partition, black, and the Block-Rectangle, gray. The vertical surface represents the value at which a Square-Corner partition becomes possible, $P_r \geq 2\sqrt{R_r}$. The black surface is valid in front of this vertical surface. The x-axis represents the value of R_r from 1 to 10, and the y-axis represents the value of P_r from 1 to 20. For highly heterogeneous ratios, *i.e.* small values of x and large values of y, the Square-Corner partition has a lower cost than the Block-Rectangle partition for the SCB algorithm on a fully connected network.

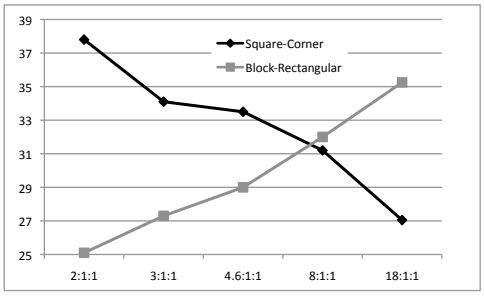

Fig. 14. Communication time in seconds for Square-Corner and Block-Rectangle partition shapes using the SCB algorithm and fully connected topology. Network bandwidth is 1000 MB/s and N = 5000. As heterogeneity increases along the x-axis the Square-Corner volume of communication decreases, eventually overtaking the Block-Rectangle partition.

has been reached. The process will then be put to sleep and woken later when the CPU has been idle long enough to achieve the desired processor speed. These results, shown in Fig. 14, confirm that the Square-Corner partition shape has a lower communication and execution time when the ratios are highly heterogeneous when compared with the Block-Rectangle partition shape.

XI. Conclusion

In this paper we expanded the Push operation to be applicable to three heterogenous processors. We described performance models for three processor execution of MMM under five different algorithms. We showed that the Push operation decreases volume of communication, and thereby total execution time for each of these five algorithms. We used an experimental technique to support our postulate that no arbitrary partition shape or non-shape exists which is superior to those shapes in Fig. 11 and Fig. 12. Our program was able to reduce the set of possibly optimal shapes to just six shapes from the total number of possible arbitrary arrangements of elements, $\frac{N^2!}{(\in P!)\times(\in R!)\times(\in S!)}$.

Analyzing the potentially optimal data partition shapes is beyond the scope of this paper, but such analyses will be presented in the future. Each of the six partition shape types will be extensively studied for all five parallel MMM algorithms, and for all processing power ratios.

Other avenues for exploration would be the introduction of additional constraints on network topology, communication latency, or cache performance in the modeling. These factors would certainly influence which shapes were optimal, and under what conditions.

The complexity of the three processor case, as compared to two processors, makes this work an excellent starting point for four or more processors. Both the analytical methods and our experimental technique used in this paper are extensible. The ultimate aim is to determine the optimal data partitioning shape under a variety of conditions for any number of heterogeneous processors.

References

[1] A. Lastovetsky and J. Dongarra, *High-performance heterogeneous computing*. Wiley, 2009.

[2] A. Kalinov and A. Lastovetsky, "Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers," in *7th International Conference on High Performance Computing and Networking Europe (HPCN Europe'99)*, 1999.

[3] R. A. Van De Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," *Concurrency: Practice and Experience*, vol. 9, pp. 225–274, 1997.

[4] O. Beaumont, V. Boudet, A. Legrand, F. Rastello, and Y. Robert, "Heterogeneous matrix-matrix multiplication or partitioning a square into rectangles: Np-completeness and approximation algorithms," in *PDP 2001*, pp. 298 –305.

[5] A. Kalinov and A. Lastovetsky, "Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers," *Journal of Parallel and Distributed Computing*, vol. 61, pp. 520–535, 2001.

[6] E. Dovolnov, A. Kalinov, and S. Klimov, "Natural block data decomposition for heterogeneous clusters," in *IPDPS 2003*, April 2003.

[7] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Matrix-matrix multiplication on heterogeneous platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, pp. 1033–1051, 2001.

[8] A. DeFlumere, A. Lastovetsky, and B. A. Becker, "Partitioning for parallel matrix-matrix multiplication with heterogeneous processors: The optimal solution," in *21st International Heterogeneity in Computing Workshop (HCW 2012)*. IEEE, 2012, pp. 125–139.

[9] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on heterogeneous multicore and multi-gpu systems using functional performance models of data-parallel applications," in *14th IEEE International Conference on Cluster Computing*. IEEE, 2012, pp. 191–199.

[10] B. A. Becker and A. Lastovetsky, "Towards data partitioning for parallel computing on three interconnected clusters," in *6th International Symposium on Parallel and Distributed Computing (ISPDC 2007)*, 2007.

[11] B. A. Becker, "High-level data partitioning for parallel computing on heterogeneous hierarchical computational platforms," PhD Thesis, University College Dublin, Dublin, Ireland, April 2011.

[12] R. Hockney, "The communication challenge for mpp: Intel paragon and meiko cs-2," *Parallel Computing*, vol. 20, no. 3, pp. 389–398, 1994.

[13] L. S. Blackford, J. Choi *et al.*, *ScaLAPACK Users' Guide*. Philadelphia, PA: SIAM, 1997.

[14] A. DeFlumere and A. Lastovetsky, "Theoretical results on optimal partitioning for matrix-matrix multiplication with two processors," School of Computer Science and Informatics, University College Dublin, Tech. Rep. UCD-CSI-2011-09, September 2011.

[15] E. Gabriel, G. E. Fagg *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *EuroPVM/MPI 2004*. Springer, 2004, pp. 97–104.

[16] R. Whaley and J. Dongarra, "Automatically tuned linear algebra software," in *ACM/IEEE Conference on Supercomputing*, 1998, pp. 1–27.