Adding Data-movement Constructs to the PGAS Parallel Computing Model

Leonardo Uribe
Vrije Universiteit Brussel - VUB
Pleinlaan 2, 1050
Brussels, Belgium
luribeka@vub.ac.be

Pascal Costanza
Vrije Universiteit Brussel - VUB
Pleinlaan 2, 1050
Brussels, Belgium
pascal.costanza@vub.ac.be

Theo D'Hondt
Vrije Universiteit Brussel - VUB
Pleinlaan 2, 1050
Brussels, Belgium
tjdhondt@vub.ac.be

ABSTRACT

Data movement is a key concept in parallel computing. Given that data movement is generally slower than data processing, it is important to guarantee an efficient use of the communication facilities of the hardware. The MPI communication protocol is the most common model to allocate and move data in High Performance Computing. MPI is a very expressive model but is also a very low-level one. More high-level models such as the PGAS (Partitioned Global Address Space) model allow easier abstraction but limit expressivity.

In this paper we present an extension of the PGAS model to include more expressivity while conserving the abstract representations of the model. Specifically, we use the concept of data shape to represent data distribution and use a generalized concept of function to move and process data.

Categories and Subject Descriptors

E.1 [**Data Structures**]: [Distributed data structures]; D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures, polymorphism*

General Terms

Algorithms, Design, Languages

Keywords

Parallel Computing, Data movement, PGAS Model

1. INTRODUCTION

A common characteristic of most of today's supercomputers is a physically distributed memory. Nowadays, we count on more processing units but also increased needs for data movement among them. Given that data movement is a expensive operation compared to data processing [6], we have to be able to allocate and move data in a smart way in order to exploit the high performance possibilities that parallel computers offer.

ECOOP '2010 Maribor, Slovenia EU

The most common way to move data in High Performance Computing nowadays is making use of the Message Passing Interface (MPI) communications protocol. MPI is basically a standard protocol specifying a communication model among processes running in a distributed memory system. MPI applications can achieve high performance, scalability, and portability. Nevertheless, MPI exposes the programmer to very low-level details of the communication process, making it hard to manage large computations and to reason about them in an abstract way.

A recent alternative to the MPI model is the Partitioned Global Address Space model. The PGAS model is a high level approach to deal with communication processes in parallel computing. It offers the programmer a model of a general shared memory and automatically manages the movement of data among real memories when needed, making it transparent for the user. The programmer thinks in terms of a single memory, but relies on some language constructs to logically divide it and thus model the real hardware memory units. This way, s/he can be aware of the movement of data between different memories while still working with a shared memory model. The PGAS implementation then matches the user specified partition with the real hardware distribution and generates all the needed communication code.

The main advantage of PGAS over MPI is that it frees the user of lot of technical communication details and allows her/him to think more clearly about the general algorithm itself. Its main disadvantage, with respect to data movement, is that it does not have the same level of expressivity as the MPI model.

The purpose of this paper is to offer an extension of the PGAS data-distribution model in order to include basic data movement operations. We introduce the concept of shape for arrays to simulate its data distribution over the hardware memories. Then, we extend the concept of function by allowing it to operate not only over the entries of the array, but also over its shape. This kind of generalized functions has the ability to move data inside these shaped objects, in the frame of their shapes.

2. THE PGAS MODEL

The increasing use of computers having multi-processing capabilities gives rise to a lot of new issues in the computing world. We were used to easily ignore the hardware architecture of our given machine and program it in an abstract way, worrying only about the problems that arise at the upper levels. Now, we have to face a variety of computer

architectures, with varying number of processors, memory distributions and communication facilities. Issues such as scalability of our program (with respect to the number of processing units), data-distribution and data movement become key concepts that have to be carefully taken into account if we want to produce efficient programs.

Nowadays, the disparity between processor and memory speed is the primary obstacle to improved computer system performance [6]. Interpreting communication as an access to an external memory, we conclude that there is a big difference in time consumption between the processing and the communication of data. Given this key fact, we have to think about data communication as one of the most important concerns for parallel programming. This problem is now being tackled with different approaches. Ranging from totally automatized [2] to totally programmer-specified [7], several models look for new data-communication constructs to write efficient, scalable and portable code.

The Partitioned Global Address Space (PGAS) Model is an approach to high-level data and task distribution based on the idea of a shared memory model plus a facility to make a logical partition of that memory. This partition allows the programmer to model the real memory distribution. The programmer can then be aware of the data movement between partitions, but s/he works over a shared-memory space and does not have to take care of the communication process. The details of the real data movement among processing units is hidden from the user and automatically realized by the compiler and run-time system.

In this section, we will describe a basic model for the PGAS data-distribution process. In the following sections, we will apply some generalizations to the model in order to include data-movement in it. We will work with Chapel as a reference of a language using the PGAS model. Chapel is a new parallel programming language that is under development at Cray Inc. in the context of the DARPA High Productivity Computing Systems initiative [4]. To build our alternative model, we will use Common Lisp, which we consider a good language for making new prototypes.

To begin, let's analyze a simple program written in Chapel. The example is taken from [9]. Figure 1 shows the Chapel code for multiplying a matrix and a vector. No data distribution is specified yet.

```
1 var A: [1..m,1..n] real;
2 var x: [1..n] real;
3 var y: [1..m] real;
4 y = sum reduce (dim=2) forall (i,j) in [1..m,1..n] A(i,j)*x(j);
```

Figure 1: Matrix-vector multiplication in Chapel without specific data-distribution directives

The first three lines define the matrix A, the vector x and the resulting vector y. All the processing is done in the fourth line. The **forall** instruction in Chapel is a variant of the standard for loop that allows for the concurrent execution of the loop body [4]. The **sum reduce** instruction adds all the elements in each row of the matrix returned by **forall**. The final result is then assigned to the variable y.

The data distribution specification for the given example is shown in Figure 2. For this example, it is supposed that we want to run our matrix-vector multiplication code on a Cell processor [8]. Roughly speaking, the cell processor is a

```
1 const SPE:[1..8] locale;
2 var A: [1..m,1..n] real  distributed(block,*) on SPE;
3 var x: [1..n] real  replicated on SPE;
4 var y: [1..m] real  distributed(block) on SPE;
5 y = sum reduce (dim=2) forall (i,j) in [1..m,1..n] A(i,j)*x(j);
```

Figure 2: Matrix-vector multiplication in Chapel with specific data-distribution directives

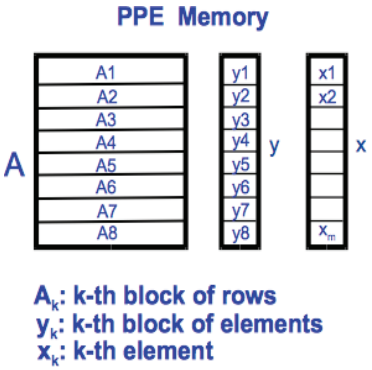

Figure 3: Data-distribution for the matrix-vector multiplication example in Chapel

multi-core processor containing a main processor called the Power Processing Element (PPE) and eight co-processors called the Synergistic Processing Elements (SPEs). Our code is supposed to run in the PPE and makes use of the 8 SPEs to parallelize the computation. The code in *italics* corresponds to the added data-distribution directives. The first line of code defines an array of type locale, which is the user representation of the set of available processing units. A processing unit is a portion of the target parallel architecture that has processing and storage capabilities [4]. The way in which the arrays are distributed is specified in lines 2-4. In line 2, the instruction **distributed** means that the array contents are going to be distributed among the available locales (elements of the locale array **SPE**), the argument **block** represents a uniform distribution of the first dimension of the array over the given locales. The argument ***** indicates that no distribution is made over the second dimension. The word **on** specifies the locale where the array is going to be distributed, **SPE** for this case. The instruction **replicated** in line 3 indicates that a copy of the array is going to be placed in each locale. Finally, in line 5 the execution of the computation takes place. A visual representation of the partitioned data is shown in Figure 3.

Once the data-distribution is made, the user is aware of the possible movements of data between the different locales, so s/he can try to avoid excessive data movement by changing the algorithm or the original data distribution. In fact, for this example, data is distributed in such a way that no extra data-movement is needed. Figure 4 shows the data used for a single processing unit. Observe that all that data is local to that SPE.

Summarizing, the PGAS model allows the user to specify the way in which data is distributed over the available processing units, making her/him aware of the possible data-

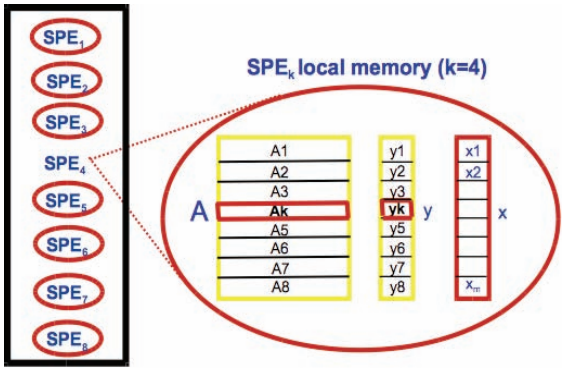

Figure 4: Data involved in each processor for the matrix-vector multiplication example in Chapel

communication processes of an algorithm. At the same time, the model offers a shared-memory model, allowing the user to reference memory positions in a simple way and automating the data transfer between physical memories when needed.

If we consider a more complex example, for instance a matrix-matrix multiplication process, we realize that not only do we have to distribute data at the beginning, but we should move data as part of the whole computation. Suppose that we are working with the Cell processor and distribute the matrices **A** and **X** as **blocks** in the first dimension as in Figure 5. Then, the computation of the elements of the resulting matrix (**Y** = **A** x **X**) located in the diagonal blocks (Y_{44}, for instance) do not involve data movement given that they are evaluated with data local to the same SPE (the 4th for the Y_{44} block). Nevertheless, elements such as the ones contained in the Y_{62} block of the resulting matrix Y do need data movement between **SPE 2** and **SPE 6** in the Cell processor. As it is not easy to determine which initial data configuration would imply the least amount of data transfers among memories, the PGAS distribution model is not sufficient to model the problem efficiently. As a consequence, the user loses control over the data movement process, which will be automatically done for the Chapel compiler, and it is not clear any more what the communication costs of the computation are.

An option to redefine the distribution of data at any moment in the program and not only in the variable definition statements is still possible. High Performance Fortran (HPF) offers that possibility [5]. Nevertheless, simulating data movement by describing successive distributions of it is not only very expensive in terms of computational time, but could be also conceptually difficult to understand. To solve this problem, we propose an enriched PGAS model including data movement constructs.

3. SIMULATING THE PGAS MODEL

Before describing the extensions of our model, let us first see how the matrix-vector multiplication looks like in it. Figure 6 shows the code for the matrix-vector-multiplication function written in Common Lisp. The arguments are non-distributed arrays.

Line 1 defines the function and its arguments. The result is passed as an argument, allowing us to determine its shape and have control over the movement of data in the computation. Lines 2-3 assign labels to the dimensions of the matrix. Lines 4-8 execute the computation.

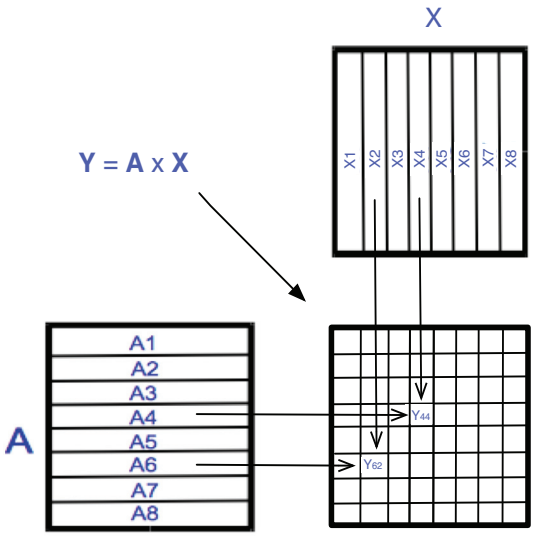

Figure 5: Data distribution example for a matrix multiplication process in Chapel using the Cell processor

```
1 (defun matrix-vector-multiply (matrix vector result)
2   (let ((rows (array-dimension matrix 0))
3         (cols (array-dimension matrix 1)))
4     (loop for row from 0 to (1- rows) do
5           (setf (aref result row)
6                 (loop for col from 0 to (1- cols) sum
7                       (* (aref matrix row col)
8                          (aref vector col)))))
9     result))
```

Figure 6: Matrix-vector multiplication function in Common Lisp with non-distributed arguments

The matrix-vector multiplication algorithm including data-distribution directives is shown in Figure 7. The code in *italics* represents the specific data-distribution directives of the model.

```
1 (setq SPEs (make-virtual-processor-array 8))
2 (setq matrix (make-array '(m n) :shape blocks :H-tiles t :on SPEs))
3 (setq vector (make-array n :shape replicated :on SPEs))
4 (setq result (make-array n :shape blocks :V-tiles t :on SPEs))
5 (matrix-vector-multiply matrix vector result)
```

Figure 7: Matrix-vector multiplication in Common Lisp with specific data-distribution directives

Line 1 defines an array of virtual processors, which are conceptually the same as the locale arrays for Chapel. Lines 2-4 define the variables to be used. Line 5 executes the multiplication process. The specific data-distribution code

(in *italics*) is optional and can be omitted without changing the semantics of the algorithm. To achieve this behavior, we have overridden the **make-array** function (lines 2-4). Each time an array is created, it is wrapped by an instance of a *shape class*, which include attributes such as **contents**, **shape**, **total-size** and **on**. If the additional attribute **shape** is omitted, only the requested array is returned. If an argument for **shape** is specified, an instance of the argument class is created and returned. Figure 8 shows the class definition for the general shape class (**shape-class**) and for its subclass **blocks**

```
1 (defclass Shape-class ()
2   ((contents :accessor contents)
3    (total-size :accessor total-size)
4    (on :accessor virtual-processors)))
5
6 (defclass Blocks (shape-class)
7   ((number-of-blocks)
8    (h-tiles :initform 1)
9    (v-tiles :initform 1)))
```

Figure 8: Definition of Shape-class and one of its subclasses

Lines 1-4 define the **Shape-class**. The attribute **contents** saves a list with the parts of each instance. The parts can be arrays or simply other shaped objects. For the matrix-vector multiplication case, **contents** for the instances of class **Blocks** store 8 arrays resulting from the partition of the original arrays (**matrix**, **result**) according to the number of virtual processors (8 for this case). The attribute **total-size** stores the number of single elements in the shaped object, that is the sum of all the elements of each entry of **contents**. This represents the size of the shared memory of the shaped object. The attribute **on** stores the **virtual-processor-array** that is associated with the shaped object and that will give the compiler the information to distribute the **contents** of the object in the available hardware. Lines 6-9 define **Blocks** as a sub-class of **Shape-class**. The new attributes simply describe some specific characteristics of this kind of shape. Whenever the function make-array is called with a **shape** specification, a shaped object is created and a split function is called to split the given array according to the given shape. Nevertheless, the user has also the possibility to create shaped objects from other shaped objects or from references to existing arrays. This is important when working with very large structures that do not fit in a single memory, like those used for quantum computing simulators [3].

Access to the shared memory is achieved by delegation. First, the shaped structure receives an index to an entry in the shared memory. Then, the partition of the contents array where the reference entry can be found is selected and the index is converted accordingly. Finally, the accessor is called recursively with the partition and the new index. Figure 9 shows a graphical example of the memory access process in a shaped object.

4. EXTENDING THE PGAS MODEL WITH DATA-MOVEMENT DIRECTIVES

In Section 3, to get a global memory element, we made use

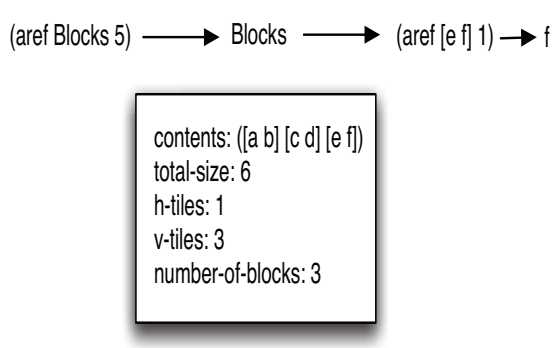

Figure 9: Example of memory access in a shaped object

of two important concepts: polymorphism and delegation. Looking at the function **aref** in Figure 9, observe that when the function is applied over a shaped object, then a process of transformation of the index is realized and the **aref** function is again applied over a new object (delegation). When the function is applied over an array, it behaves differently (polymorphism), retrieving the element at the given index.

By extending the behavior of the **aref** function to any other function, we can generalize the concept of function: a function can act over data but can also act over the shape of that data (wrapping shaped object). When the function acts over data, it transforms the data and we have the kind of functions we are familiar with. When the function acts over the shape of the data, we can think about new transformations such as changing data position in the structure and selecting to which parts of the structure to delegate the function.

To see how this new generalized functions can be used, let us work with a simple example: square matrix transpose. Figure 10 shows the code for a function that transposes a given matrix. The arguments are non-distributed arrays.

```
1 (defmethod transpose (2d-array result)
2   (let* ((rows (array-dimension 2d-array 0))
3          (cols (array-dimension 2d-array 1))
4     (dotimes (i rows)
5       (dotimes (j cols)
6         (setf (aref result j i) (aref 2d-array i j))))))
```

Figure 10: Function to transpose a matrix in Common Lisp with non-distributed arguments

Line 1 defines the function and its arguments. We have defined a method instead of a function because we want to use polymorphism over the arguments. Lines 2-3 define some local variables and lines 4-7 do the transpose operation.

An alternative way of transposing a square matrix is representing it as a quad-tree shaped object and successively swapping its leaves [1]. A quad-tree represents a tree structure in which each node has 4 child nodes. An example of a quad-tree representation of a square matrix is depicted in Figure 11. We begin with a 4x4 matrix, make an intermediate representation of a quad-tree consisting of the root

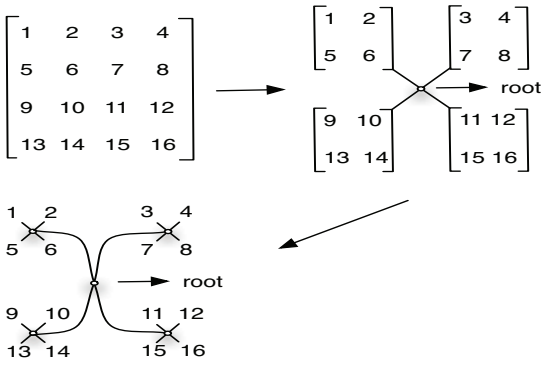

Figure 11: Representation of a matrix as a nested quad-tree

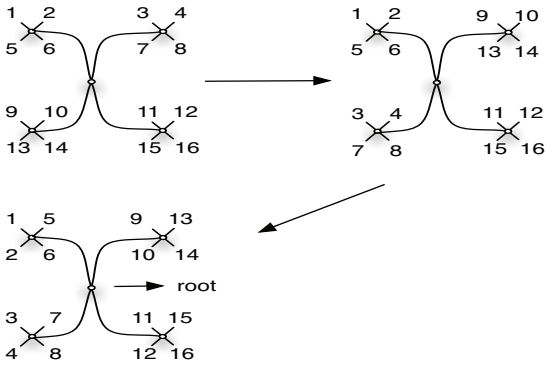

Figure 12: Transpose algorithm for a matrix represented as a nested quad-tree

and four 2x2-matrix leaves and finally represent it as a two-level quad-tree whose leaves are the elements of the original matrix.

An informal description of the transpose algorithm for a square matrix represented as a nested quad-tree is: swap the upper-right and the lower-left child nodes of the root node and repeat the procedure recursively for each sub-node. Figure 12 graphically shows this idea.

In general, a square matrix can be transposed using a combination of the quad-tree method and the classical method. First, we begin swapping nodes until we reach the nodes whose dimension we consider appropriate to apply the standard method. The advantages and disadvantages of each method depend, among other factors, on communication vs. processing speeds of the computer in use.

Figure 13 shows the code to transpose a square matrix using our PGAS extended model.

The code in *italics* represents the added directives and can be omitted if no distribution is specified. Lines 1-2 create a squared **matrix** and specify that it is distributed as a **quad-tree**. Lines 3-4 do the same for the variable **result**. The **:max-size** instruction indicates the maximum allowed size for the leaves of the **quad-tree**. Then the original matrix is split until the number of elements in each leaf is **max** or less. The **:on** instruction specifies the array of virtual processors

```
1 (setq matrix (make-array '(n n) :shape quad-tree
2                          :max-size max :on VP))
3 (setq result (make-array '(n n) :shape quad-tree
4                          :max-size max :on VP))
5 (defmethod transpose ((matrix quad-tree) (result quad-tree))
6   (swap-tiles matrix 2 3 result)
7   (continue-with #'transpose result result))
8 (transpose matrix result)
```

Figure 13: Matrix transpose using our PGAS extended model

where the object is distributed. The specification of **VP** is not very important for this example and is supposed to be previously defined. Lines 5-7 define the transpose operation for **quad-trees**. The method transpose is polymorphic and its definition for 2-dimension arrays is already given in Figure 10. Line 6 swaps (**swap-tile**) two partitions of the **contents** of **matrix** and assigns the result to **result**. The swapped partitions are, according to Figure 12, the upper-right and the lower-left of the **quad-tree**. Line 7 executes the instruction **continue-with**, which applies the transpose function to all of the nodes in the next level of the **quad-tree**. Eventually, the function will reach the leaves and the standard transpose function will be applied.

At this point, it is important to make a remark about parallelism. In the original PGAS model, the parallelism tends to be orthogonal to the data distribution. Looking at the code of Figure 2, we can see that the parallel execution of the process is dictated for the **forall** instruction and maybe for the **sum-reduce** instruction, but not for the data-distribution directives. For our model, the same is true when the only functions interacting with the shaped object are the getters and setters. For the more general case of other functions which interact not only with specific entries of the shaped object but with more complex sets of data inside, the fine granularity of the parallelism is not guaranteed by the usual language constructs and has to be expressed and implemented as part of the model. The ideal place for it could be the **continue-with** method (see Figure 13), which passes the function from the shaped object to its parts. This instruction is implemented with the Lisp instruction map, that applies a function to all of the elements of a list (the **contents** attribute in this case).

5. CONCLUSIONS

The PGAS model for distributed parallel computing offers to the programmer all the facilities of a shared-memory model and also the constructs to make logical partitions of the memory which allows simulation of the real distributed memories. The programmer does not have to worry about data communication processes, but can be aware of the amount of data communication in her/his algorithm via the logical partitions.

The PGAS model is however limited in the sense that it has data allocation directives but lacks data movement operations. In this paper we propose an extension of the model to allow the expression of data movements.

We introduce the concept of shape for arrays to simulate their hardware distribution and extend the concept of function to interact not only with the data but also with the shape of the data. In this way we allow constructs inside the function to move data inside a shaped object and also

to determine the way in which it is applied to the partitions of the object.

The actual state of the model allows the swapping of partitions of the shaped object as well as single elements. By now, the only way to pass a function to the next structural level is by applying it to its partitions. We have implemented the basic matrix-matrix operations (multiplication, sum, transpose) and we work on the implementation of the Fast Fourier Transform as an example.

As future work, we consider the possibility to extend the shape-related constructs further to allow a function to change the shape of the structures and show more complex behavior when interacting with shaped-objects.

6. REFERENCES

[1] S. K. Abdali and D. S. Wise. Experiments with quadtree representation of matrices. In *ISAAC '88: Proceedings of the International Symposium ISSAC'88 on Symbolic and Algebraic Computation*, pages 96–108, London, UK, 1989. Springer-Verlag.

[2] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan. Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 1–10, New York, NY, USA, 2008. ACM.

[3] K. De Raedt, K. Michielsen, H. De Raedt, B. Trieu, G. Arnold, M. Richter, T. Lippert, H. Watanabe, and N. Ito. Massive parallel quantum computer simulator, 2006.

[4] C. Inc. *Chapel Language Specification 0.785*. 901 Fifth Avenue, Suite 1000, Seattle, WA 98164.

[5] C. Inc. *High Performance Fortran Language Specification 2.0*.

[6] N. R. Mahapatra and B. Venkatrao. The processor-memory bottleneck: problems and solutions. *Crossroads*, page 2.

[7] G. W. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. MIT Press, Cambridge, MA, USA, 1989.

[8] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.

[9] H. P. Zima. Advanced programming and execution models for future multi-core systems, September 2007.