Single Operation Multiple Data - Data Parallelism at Subroutine Level

Eduardo Marques  
CITI / Departamento de Informática  
Faculdade de Ciências e Tecnologia  
Universidade Nova de Lisboa  
2829-516 Caparica, Portugal

Hervé Paulino  
CITI / Departamento de Informática  
Faculdade de Ciências Le Tecnologica  
Universidade Nova de Lisboa  
2829-516 Caparica, Portugal  
Email: herve@di.fct.unl.pt

Abstract—The parallel nature of the multi-core architectural design can only be fully exploited by concurrent applications. This status quo pushed productivity to the forefront of the language design concerns. The community is demanding for new solutions in the design, compilation, and implementation of concurrent languages, making this research area one of great importance and impact. To that extent this paper proposes the expression of data parallelism at subroutine level. The calling of a subroutine in this context spawns several execution flows, each operating on distinct partitions of the input dataset. Such computations can be expressed by simply annotating sequential subroutines with data distribution and reduction policies, delegating the management of the parallel execution to a dedicated runtime system. The paper overviews the key concepts of the model, illustrating them with some small programming examples, and describes a Java implementation built on top of the X10 [1] runtime system. A performance evaluation attests that this approach can provide good performance gains without burdening the programmer with the writing of specialized code.

Keywords— Concurrent Programming, Data Parallelism, Single Operation - Multiple Data

I. INTRODUCTION

The shift from frequency to core scaling in CPU design ushered a new era in computer organization. Concurrent programming has been under the spotlight ever since, considering that the increase of the number of cores does not imply faster sequential applications. Nonetheless, although multi-core architectures have been the standard processor architectural design for over six years now, the mainstream programming languages, compilers and runtime systems remain mostly unaltered.

The community is demanding for new solutions in the design and implementation of concurrent languages in general, making this research area one of great importance and impact. Such has been recognized by both academia and industry, fostering the proposal of several new programming languages, such as X10 [1], Chapel [2], Fortress [3] and Axum [4], and libraries, such as Intel’s Parallel Building Blocks suite [5] and Java’s Fork/Join framework [6].

The uttermost objective is the transition of parallel computing from the scientific community to the mainstream of software development. This requires the definition of linguistic constructions that, on one hand, permit the simple and natural expression of parallelism, hence not degrading the level of productivity currently observed in general purpose languages, while on the other, do not compromise performance nor restrict the possibility of tuning the code, when needed.

Task parallelism is often explored by providing the means for subroutines (functions, methods, service operations, and so on) to be offloaded to a pool of worker threads, working on behalf of the application. Cilk [7], X10 and Java’s executors, among others, built from this concept. Whereas, data parallelism is usually explored at loop-level, of which OpenMP [8], Intel TBB [9] and PGAS languages, such as UPC [10] and also X10, are examples.

This paper proposes the expression of data parallelism at subroutine level. The calling of a subroutine in this context spawns several tasks, each operating on a different partition of the input dataset. These tasks are offloaded for parallel execution by multiple worker threads, and run in conformity to a variation of the Single Program Multiple Data (SPMD) execution model, which we have baptised as Single Operation Multiple Data (SOMD).

The execution model is presented to the programmer as a Distribute-Map-Reduce paradigm: the input dataset is decomposed so that multiple instances of the subroutine can be executed in parallel (the map stage), and their partial results reduced to compute the final result.

This approach provides a framework for the average programmer to express data parallel computations by annotating unaltered sequential subroutines, hence taking advantage of the parallel nature of the target hardware without having to program specialized code. We envision that commonly used distributions and reductions will be available in the form of libraries. In such cases, the programmer’s work can be confined to the selection of the distribution and reduction policies to be applied. This, however, does not prevent the definition of custom policies, or even the tailoring of the method’s code to better adjust it to the SOMD model, either due to performance or algorithmic issues.

We instantiated the model in the Java programming language, using the X10 Java runtime system (X10JRT) as the backbone for our own runtime system. X10JRT provides us the framework to execute Java methods complying to
our specification (SOMD methods from now on) in shared and distributed memory architectures, being that our current focus is on multi-core architectures.

The remainder of this paper is structured as follows: Section II overviews the SOMD model; Section III describes how it has been instantiated in the Java programming language; Section IV evaluates our prototype implementation from a performance perspective; Section V compares our work to others in the field, and; finally Section VI presents our concluding remarks and prospective future work.

II. THE SOMD EXECUTION MODEL

The SOMD execution model consists on carrying out multiple instances of a given method\(^1\) in parallel, over different partitions of the input data. The invocation is decoupled from the execution, a characteristic intrinsic to the active object [11] design pattern that makes the parallel nature of execution model transparent to the invoker. As is illustrated in Figure 1, the invocation is synchronous, and the execution is carried out by multiple concurrent flows. The figure also unveils the distribution and reduction stages. As will be further detailed in Section II-A, the first decomposes the input dataset in a pre-defined number of partitions, whilst the second computes the final result from the ones produced by each method execution.

Although we are mainly focussed on multi-core architectures, the model can be also be applied to distributed memory environments. The portability of the code from the former to the latter is subservient to the argument evaluation strategy of the hosting language. The passing of references to memory positions in distributed environments requires a distributed memory management layer that falls out of the scope of this paper. The absence of such layer causes each method instance (MI) to hold a local copy of the argument.

Figure 2 showcases the execution model in distributed memory environments, which follows a master-worker pattern. The node running the invoking thread assumes the master role, performing the following tasks: i) top-level application of the distribution stage; ii) dispatching of the computation to the worker nodes; iii) execution of a subset of the MIs; iv) collection of the partial results, and; v) computation of the reduction stage.

\(^1\)Our current research is applying the SOMD model to object-oriented languages, thus, from this point onward, we will use the method terminology rather than subroutine.

![Figure 1. Transparency to the caller](image1.png)

![Figure 2. Execution in heterogeneous environments](image2.png)

The distribution stage is hierarchical, the input data set is first decomposed among the target nodes and then among the workers running in each node. In turn, there is no native support for parallelism in the reduction stage. This is motivated by the facts that: 1) in single machine environments the number of partial results to reduce are usually small, hence not heavy enough to justify a parallel execution, and 2) the hierarchical application of a reduction policy may compromise the correctness of the computed results. For example, given a set of values \(v_i : i \in [1, n + m]\):

\[
\frac{\text{average}(v_1, \ldots, v_n)}{\text{average}(v_{n+1}, \ldots, v_{n+m})} \neq \frac{\text{average}(v_1, \ldots, v_n, v_{n+1}, \ldots, v_{n+m})}{\text{average}(v_{n+1}, \ldots, v_{n+m})}
\]

Therefore, instead of exposing these details to the programmer, we opt to relegate them to the reduction strategy implementation, which may itself resort to a SOMD method.

A. Distribute-Map-Reduce Paradigm

The model is presented to the programmer in the form of a Distribute-Map-Reduce (DMR) paradigm that borrows some ideas from the MapReduce framework [12]. A brief description of each stage follows:

**Distribute** - partitions the target value into a collection of values of the same type. It can be applied to multiple input arguments and local variables.

**Map** - applies a MI to each partition of the input data set.

**Reduce** - combines the partial results of the previous stage to compute the method’s final result.

Note that the method to apply does not have to be altered. Given a argument of type \(T\), a distribution over such argument must by a function of type \(T \rightarrow \text{List} < T >\). Moreover, a reduction applied to a method that returns a value of type \(R\) must be a function of type \(\text{List} < R > \rightarrow R\). Thus, the method’s application complies to its original prototype, since it receives one of the elements of the distribution set (of type \(T\)) and provides an output of type \(R\). Figure 3 depicts a simplified version of the paradigm comprising a single distributed value.

B. Constructs

The model has been instantiated as an extension to the Java programming language. The adoption of the extension approach arises from the fact that this work integrates a
larger research project that addresses a service-based concurrent programming model [13], which is being implemented as a Java extension. The concrete syntax here presented serves only as a vehicle to expose our fundamental research and is not the paper’s main focus.

1) Distributions and Reductions: We begin by presenting the base distribution/reduction constructions. Distributions are declared by qualifying a given variable with the dist modifier as follows:

\[
\text{dist } C(\vec{e})\ \text{variable_declaration}
\]

where \(C(\vec{e})\) denotes a constructor of the class implementing the distribution policy. If the variable’s type is an array, the policy may be omitted in favour of a default behaviour, which partitions the array as evenly as possible.

In turn, reductions are applied to methods. The reduce-with modifier qualifies their declaration as follows:

\[
\text{method_header reducewith } C(\vec{e}) \{ \text{method_body} \}
\]

We opt to place the modifier after the header declaration for readability reasons.

Distribution and reduction policies must, respectively, conform to the Distribution and Reduction interfaces depicted in Listing 1. Note that, in Java, the use of distributed variables implies copying the contents of the original data structure to the newly created partitions. In Section IV we will address strategies to avoid such overhead.

Listing 1. The Distribution and Reduction interfaces

Listing 2 illustrates the use of the proposed constructs in a simple addition of integer arrays. The use of the default distribution splits the input arrays as evenly as possible among the available workers. The target method applies the sequential algorithm to the adjudicated partition and produces a partial result that is assembled by the ArrayAssembler reduction (Listing 3). Note that this is a general reduction that can be used to assemble any kind of arrays, and thus suitable to be included in a library. Optimized implementations can be developed, for instance to receive the size of the final array as argument and, with that, avoid the computation of the value from the size of the partial arrays.

Listing 2. The sum of two arrays in parallel

Listing 3. The ArrayAssembler reduction

2) Shared Variables and Synchronization Constructs: The shared variable modifier enables the sharing of local variables among the multiple MIs. Up this point only non-distributed object parameters were considered to be shared.

To synchronize the access to shared variables we permit the delimitation of atomic blocks. Note that these only
synchronize the MIs and do not try to acquire the object’s monitor. For that purpose synchronized blocks should be used instead.

Furthermore, we allow for synchronization points in the form of barriers. Listing 6 exemplifies the use of all these constructs: line 2 declares a shared local variable, lines 7 to 9 modify the value of such variables in mutual exclusion, and line 10 guarantees that the variable holds its final value before it is used by all MIs.

Listing 6. Shared variables and synchronization constructs

3) Distributed Shared Arrays: Some problems greatly benefit from the overlapping of the partitions, namely to enable the reading and writing of positions beyond the distributed boundaries. When only reading operations are in order, a custom distribution strategy does the job. However, since distributed values are not shared between MIs, modifications performed by one are not seen by the remainder. To overcome this limitation we propose the notion of distributed shared array, which enables an MI to access elements adjacent to its partition. The syntax is a follows:

distshared($e_1,e_2$) variable_declaration

where $e_1$ and $e_2$ are expressions that denote the number of accessible adjacent positions to the left and the right, respectively.

Listing 7 applies the construct to the replacement of all occurrences of a string in a given text. In order to be suitable for such distribution the text must assume the form of an array of characters. The distribution allows for each MI to access as many positions to the right as the ones necessary to hold an occurrence of the string to replace (oldStr) that begins within the boundaries of the assigned partition.

Listing 7. Replacing all the occurrences of a string in a text

III. ARCHITECTURE AND IMPLEMENTATION

The model is being instantiated as an extension to the Java language. The new constructs are translated into Java by a dedicated compiler built on top of Polyglot [14], a Java-to-Java compiler that provides a framework to easily extend the language. This compiler guarantees the program’s type safety and generates Java code, which is in turn compiled by a standard Java compiler.

The runtime system is based on the X10 Java runtime system (X10JRT). X10 [1] is a concurrent programming language that implements a Asynchronous Partitioned Global Address Space (APGAS) model [15]. Computation happens at places, abstractions for a portion of the partitioned address space plus a set of asynchronous computations (referred to as activities) that operate on such portion. The language can be compiled either to C++ or Java. We leverage on the latter’s runtime system to be able to distribute work within and across multi-core architectures.

A. Compilation into the X10 Java Runtime

Distributions generate arrays of the type to distribute. We experimented with both Java native and X10 arrays. The fact that the remainder of the application operates on Java types led us to choose the former, since the use of X10 arrays would impose the overhead of runtime type conversion.

Naturally, the MIs of a SOMD method are mapped into X10 activities. Their execution is delimited by a finish block, a synchronization barrier that ensures the termination of the spawned MIs before the execution of the reduction stage.

Shared variables are mapped into X10 shared variables, GlobalRefs, whose scope may encompass several places. Both communication and consistency concerns are delegated to the X10JRT. Figure 4 illustrates the mapping of a method that distributes three arrays (two parameters and a local variable) and resorts to a shared variable.

Atomic blocks are mapped in their X10 counterpart, while barriers are mapped into clocks, a construct that extends the notion of split-phased barrier to allow multiple activities to register and synchronize in multiple points. Activities can be associated to a clock through the clocked keyword as follows: async clocked($cc$) { code } where $cc$ denotes a sequence of previously declared clocks. Each barrier construct is mapped into a call advance() statement, which causes the activity to block until all the remainder registered in clock $cc$ have reached the same synchronization point.

Finally, distributed shared arrays are mapped into X10 DistArrays, which are distributed along multiple places. Every array access triggers a verification that checks if the
given index is mapped onto the current place. If not, the computation is shifted to the place that holds such position, so that the write or read operation is performed locally (X10 activities cannot access data located at a place other than their own). This solution assures that all overlapping positions are shared between the MIs.

B. The Runtime System

The runtime system resorts to the X10JRT to support the execution of SOMD methods within and across multicore machines. A dedicated thread is spawned to launch the X10JRT, create a work-queue that establishes the communication bridge between both runtimes, and wait for execution requests. In turn, a SOMD method invocation from a Java application thread enqueues the request in the cited queue, and blocks on a condition variable until the result has been computed - conforming to Java’s synchronous call semantics.

Request processing is serialized, i.e. only one SOMD method executes at a time. Each request generates as many MIs as workers in the X10 thread pool, thus exhausting the available resources. Future work will agile the configuration of the system so that multiple SOMD methods may execute concurrently. Figure 5 illustrates the integration of both runtime systems.

IV. Evaluation

This section evaluates of implementation from a performance perspective. Our analysis has a twofold character: first, it is a speedup analysis of the SOMD to Java methods; secondly it is a comparative analysis of the SOMD and X10 implementations of a set of benchmark applications. All the applications used in this evaluation are available at http://asc.di.fct.unl.pt/serco/somd.

All measurements were performed on a system composed of two Quad-Core AMD Opteron 2376 CPUs at 2.3 GHz and 16 Gigabytes of main memory, running the 2.6.26-2 version of the Linux operating system.

A. Speedup Analysis

This study comprised many SOMD method implementations, namely arithmetic and statistic operations, and sorting algorithms. This presentation focuses the most representative: the calculation of an histogram and of the mean value of an array, the merge-sort algorithm, the K-means algorithm and two implementations of matrix multiplication: lines distributes the lines of the first matrix, while cols distributes the columns of the second.

We experimented with array sizes of different orders of magnitude. For a simpler reading of the results we classify these in classes A to D. Table I presents the configuration settings, the execution times of the original Java methods, and the partitioning overhead for each class.

The graphs depicted in Figure 6 present the speedups obtained for each application in relation to the original sequential implementation. These measurements account both the execution of the method and the time spent in the work-queue. The latter is however negligible since our benchmark infrastructure runs a single method at a time. We omit the graphs for class D since the matrix multiplication examples are the only to obtain very timid speedups.

Graphs (a) to (c), in particular, present the speedups for the direct application of the DMR paradigm. Direct application indicates that the original methods were not modified in order for the paradigm to be applied. We will refer to this approach as partition-based, since the input data set in effectively partitioned and copied to a new location.

Analysing the results we can observe that the matrix multiplication implementations are the ones that perform best. This is naturally due to the computational weight of the original method. The lines implementation achieves a linear speedup for classes A and B, lowering its performance to a peak of 5.2 in class C, and a peak of 1.4 in class D, both at 8 cores. In turn, the cols implementation benefits from a nearly perfect mapping on the computer’s memory hierarchy to achieve a super-linear speedup of 14.4 and 19.7 in classes A and B, respectively. For example, in class A the execution time drops to 15.4s on the 8 core configuration.

Of the remainder, K-means is the one that performs best, achieving a interesting speedup of 6 in the class A configuration. On the other end is Mean, whose short execution time does not justify the partitioning overhead. Finally, the performance of Merge-sort is bounded by the reduction stage, which has to sort the partial arrays. This application is a good candidate for the implementation of a parallel reduction strategy.

Graphs (d) to (f) present the speedups for a different implementation strategy that will refer to these as range-based. The motivation is to reduce the partition overhead by distributing ranges of the target data structures(s) among the MIs, rather than the structure itself. This requires modifying the method’s code to introduce the notion of partial computation, since the algorithm no longer has the illusion of operating over the whole structure. Listing 8 showcases the range-based implementation of the example of Listing 2.

The performance gains are mostly notable in Mean, where
B. X10 Comparative Analysis

This second analysis has the objective of assessing the quality of our prototype implementation. For that purpose we ported a number of X10 benchmark applications to the SOMD model, and compared the results obtained by both implementations.

The applications included in the study are: Monte Carlo - the calculation of an estimate of the \( \pi \) constant through the Monte Carlo method; Mandelbrot - the calculation of the Mandelbrot set; Crypt - the encrypting and decrypting of a text, and Stream - computation of formula: \( a \leftarrow b \ast c + c \), where \( a \), \( b \) and \( c \) are arrays of integers.

All the applications but Monte Carlo distribute at least one array. In fact, using the X10 reference configurations: Crypt distributes an array of 50 Megabytes, Mandelbrot performs two distributions over a matrix of 6000 \times 750 bytes and over an array of 750 doubles, and Stream distributes three arrays of 33 Megabytes.

Figure 7 reports the overhead of the SOMD implementations in relation to their X10 counterparts. The Monte Carlo and Mandelbrot benchmarks break even, with X10 scaling a little better in the first, and SOMD in the both implementations of the second. The partition-based implementation of Crypt loses to X10, which scales better than SOMD up to the 38\% gain in the 8 worker configuration. The range-based implementation, however, outperforms X10 in 75\%. This can be explained by the number of Java operations statements generated by the X10 compiler (over 1000), whilst the code generated for the SOMD implementation uses only 60 of such statements. Finally, the SOMD partition implementation of stream benchmark suffers from the distribution of the 3 big arrays. Its performance is around 18 times worst than X10\(^2\). The range-based version greatly reduces this overhead, scaling even better than the original X10 implementation.

<table>
<thead>
<tr>
<th>Applications</th>
<th>Configuration</th>
<th>Execution time (s)</th>
<th>Partition overhead (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Histogram</td>
<td>array size: 100,000,000</td>
<td>2.34</td>
<td>0.5</td>
</tr>
<tr>
<td>Mean</td>
<td>array size: 100,000,000</td>
<td>0.21</td>
<td>0.5</td>
</tr>
<tr>
<td>Merge-sort</td>
<td>array size: 100,000,000</td>
<td>25.98</td>
<td>0.5</td>
</tr>
<tr>
<td>K-means</td>
<td>nr observations: 25,000,000</td>
<td>141.40</td>
<td>0.165</td>
</tr>
<tr>
<td>Matrix Multiplication - lines</td>
<td>matrix size: 2000</td>
<td>222.94</td>
<td>0.00007</td>
</tr>
<tr>
<td>Matrix Multiplication - cols</td>
<td>matrix size: 2000</td>
<td>222.94</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table I

REFERENCE TABLE FOR THE PROBLEM CLASSES

C. Productivity Analysis

We performed a small productivity analysis by measuring the amount of code required to implement the distribution and reduction policies for each application. Table II presents such measurements, corroborating our statement that the proposed model is simple to use. This affirmation is even more substantiated with the facts that: 1) most array manipulation methods can resort to the default distribution; 2)
V. RELATED WORK

Concurrent and parallel computing have been important research areas for a long time, and along the years many programming models have been proposed. Nonetheless, when it comes to mainstream programming, concurrency is usually obtained by explicitly creating new execution flows supported by thread APIs. On the parallel computing field, MPI [16], a message-passing communication standard, has been the community’s de-facto parallel programming model since 1994. MPI defines a library of low-level operations that provide high flexibility, allowing for the programming of both the SPMD (Single Program Multiple Data) and the MPMD (Multiple Program Multiple Data) paradigms.

Both these approaches provide a low-level of abstraction that requires the explicit management of the execution flows and of their interaction, which is error-prone and results in hard to maintain code. Moreover, when it comes to parallel algorithms, these approaches require the implementation of per-processor versions of the sequential algorithms.

A popular framework is OpenMP [8], a mix of compiler directives, library calls and environment variables that provide an easy way to annotate parallelism in a shared memory model. Data-parallelism in OpenMP is at loop-level, which contrasts with our subroutine approach.

Cilk [7] is still a reference when it comes to functional (or task) parallelism. It extends the C language with primitives to spawn and synchronize concurrent tasks (C functions) in shared-memory architectures. dipSystem [17] applied a similar approach for distributed environments providing a uniform interface for the programming of both shared and distributed memory architectures. Cilk has recently been incorporated by Intel in their PBB suite [5].

A data parallel programming model that is quite popular is PGAS (Partitioned Global Address Space), instantiated by UPC (Unified Parallel C) [10] and Titanium [18]. PGAS extends array programming, allowing execution flows to transparently communicate by accessing array positions other than the ones they own, being all communication compiler generated. As all array programming languages, PGAS languages are restricted to the SPMD paradigm, which is not flexible nor supports nested parallelism. Languages X10 [1] and Chapel [2] overcome this limitation by extending PGAS with processing unit abstractions (localities) to which code may be shipped and executed. Language constructs allow a) the asynchronous spawning of tasks in localities, b) the distribution of arrays across localities and the subsequent spawning of activities to operate over the distributed data, and c) the management of the results of such asynchronous activities. This model allows programming both MPMD and SPMD paradigms, but data parallelism is still performed at loop-level by iterating over the distributed data structures.

Other recent developments are the Intel PBB suite [5], and the Axum [4] language by Microsoft. The work of Intel focuses only on shared memory multi-core architectures. PBB incorporates Cilk with libraries to a) abstract threads by aggregating them into higher-level constructs named tasks, hence increasing productivity by hiding low-level details, and b) provide the seamless distribution of arrays among
several cores, borrowing the concepts of array programming languages. Axum is .NET based language that grows from the actor model to express parallelism.

VI. CONCLUSIONS AND FUTURE WORK

This paper proposes the expression of data parallelism at subroutine level. These can be annotated with distribution and reduction policies, in order to be executed according to a SOMD execution model. Such approach allows for non-experts in parallel programming to be able to express data parallel computations with a small learning curve.

A Java implementation allowed us to annotate a wide set of methods, and with that assess the model’s usefulness and our prototype’s performance. The evaluation presented in Section IV demonstrated that the simple annotation of distribution and reduction policies obtained speedups in many scenarios. The temporal complexity of the remainder was to low to subsume the partitioning overhead. The situation was overcome with the introduction of slight alterations on the original code, which greatly reduced the cited overhead.

Although our initial results are good, the X10 runtime system often imposes an extra overhead. Its features are very useful for distributed memory environments, but we think that we are able to improve the performance on a single Java virtual machine with a dedicated runtime. We are working in such enterprise.

Future work also includes the implementation of parallel reduction policies, the formalization of the proposed constructs, and usability tests.

ACKNOWLEDGEMENT

This work was partially funded by PEst-OE/EEI/UI0527/2011, Centro de Informática e Tecnologias da Informação (CITI/FCT/UNL) - 2011-2012.

REFERENCES


