Assessing Modern Parallel Programming Languages

Konstantina Panagiotopoulou
H00135257

August 2013

Computer Science
School of Mathematical and Computer Sciences

Dissertation submitted as part of the requirements for
the award of the degree of MSc in Advanced Internet
Applications
Declaration

I, Konstantina Panagiotopoulou, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the words of other authors in any form e.g., ideas, equations, figures, text, tables, programs etc are properly acknowledged. A list of references employed is included.

Date: .......................................

Signature: ..................................
Abstract

The growing development of high performance machines has led to the need for new software, able to fully utilize the potential of the underlying hardware (as multi-cores). A serial program, nowadays, is considered a slow program and it is unable to profit from the increasing number of cores of modern hardware. On the other hand, parallel programs are difficult to develop and tune, and this constitutes one of the main challenges for the software industry today.

Within this context, during the last decade, research has focused on the development of novel parallel languages, which will address modern hardware, and in particular multi-core processors. The Partitioned Global Address Space (PGAS) programming model is a promising new approach, introducing the idea of global view of physically distributed data. Based on this principle, a new “family” of languages has emerged, referred to as PGAS languages.

This project performs a detailed assessment of three out of five languages of the PGAS family, Chapel by Cray Inc., X10 by IBM Research and Unified Parallel C (UPC) by the University of California, Berkeley. The assessment covers programmability as well as performance. It addresses issues of technical features, usability of development tools and current community work on the languages. In order to assess performance, we implement three versions of the same algorithm and compare their performance on distributed and shared memory configurations. The application selected is the N-body problem, a data intensive calculative problem used in astrophysics and molecular biology. The algorithm used is the all-pairs algorithm, which requires full data exchange and therefore represents a stress-test for the communication support of the languages.
Acknowledgements

Firstly, I wish to thank my supervisor Dr. Hans-Wolfgang Loidl for his support and assistance throughout this project. His technical expertise and programming experience have helped me overcome problems in all parts of this work. Moreover, his continuous interest in my progress and his planning of regular meetings has been of crucial importance to follow the initial time planning. Finally, his guidance towards relevant scientific research and his persistence on abstraction, gave me the foundation for a deeper understanding of issues, in the field of High Performance Computing.

Furthermore, I would like to thank Josh Milthrope and Vijay Saraswat from X10’s developer team, for pointing me to the implementation on which I based my final version of the code and Mikio Takeuchi for his help with the SumReducer module. Also, Brad Chamberlain, from Chapel, for his willingness to answer my enquiries concerning Chapel’s execution model and for reviewing my code.

I am also grateful to all my lecturers, who’s teaching made this course interesting and intense. Furthermore, I would like to thank my fellow classmates for their friendship, their cooperation and their willingness to introduce me to their cultures.

Lastly, I need to thank my family and friends, back in Greece, for their support and their love throughout this challenging year away from home.
## Table of Contents

Declaration .......................................................................................................................... 2
Abstract ............................................................................................................................... 3
Acknowledgements ........................................................................................................... 4
Table of Contents ............................................................................................................... 5
List of Figures ..................................................................................................................... 8

1 Introduction .................................................................................................................. 10
   1.1 Background ............................................................................................................. 10
   1.2 Aims & Objectives ............................................................................................... 10
   1.3 Requirements Analysis & Methodology ............................................................... 12
      1.3.1 Small Scale .................................................................................................... 12
      1.3.2 Large Scale .................................................................................................... 13
   1.4 Professional, Legal, Ethical and Social Issues .................................................... 13
   1.5 Structure ............................................................................................................... 14

2 Literature Review ........................................................................................................ 15
   2.1 The evolution of machines .................................................................................... 15
   2.2 The need for parallelism ....................................................................................... 15
   2.3 Parallel Architectures ........................................................................................... 16
      2.3.1 Memory models ............................................................................................. 16
      2.3.2 Programming models ..................................................................................... 18
      2.3.3 Widely Used Parallel Languages/ Libraries ................................................... 20
   2.4 Chapel, X10 & UPC .............................................................................................. 21
      2.4.1 Development motivation ............................................................................... 21
      2.4.2 Requested features ....................................................................................... 21
      2.4.3 The PGAS family of languages .................................................................... 22
      2.4.4 Language characteristics ............................................................................. 22
   2.5 Designing Parallel Algorithms .............................................................................. 33
      2.5.1 Partition ........................................................................................................ 33
      2.5.2 Communication .............................................................................................. 33
2. X10 ................................................................. 94
3. UPC ................................................................. 104
Additional Graphs ....................................................... 111
List of Figures

Figure 2.A Moore’s Law .................................................................................................................. 15
Figure 2.B A shared memory system [57] .................................................................................... 16
Figure 2.C A distributed memory system[5] .................................................................................. 17
Figure 2.D The Map Reduce skeleton [11] .................................................................................. 19
Figure 2.E UPC pointers[34] ........................................................................................................ 31
Figure 2.F A two dimensional simulation of two colliding galaxies [36] ................................. 35
Figure 2.G Forces applied to the object m1 Figure 2.H Calculation of the force received by m2 .... 36
Figure 3.A Chapel - User mailing list activity .............................................................................. 40
Figure 3.B X10 - User Mailling list activity ................................................................................ 43
Figure 4.A Block-distributed domain (8*8) on 6 locales .............................................................. 47
Figure 4.B Example of vector addition using blocked layout[34] ................................................. 48
Figure 4.C Overview of the algorithm (communication & computation) .................................... 52
Figure 4.D Chapel runtime graph .................................................................................................. 63
Figure 4.E Chapel speedup graph .................................................................................................. 64
Figure 4.F X10 runtime graph –Distributed Memory .................................................................. 65
Figure 4.G X10 speedup graph – Distributed Memory ................................................................. 66
Figure 4.H UPC runtime graph – Distributed Memory ................................................................. 67
Figure 4.I UPC speedup graph – Distributed Memory .................................................................. 68
Figure 4.J Speedup comparison for 24000 bodies input –X10 ...................................................... 73
Figure 4.K Speedup comparison for 24000 bodies input –UPC .................................................... 74
Figure 4.L Comparative Speedup for 24000 bodies - Distributed Memory ................................. 75
Figure 4.M Comparative Speedup for 24000 bodies - Shared Memory ...................................... 76
Figure 0.A Chapel - Absolute speedup graph?? .......................................................................... 111
Figure 0.B X10 relative speedup graph ...................................................................................... 112
Figure 0.C X10 runtimes for larger inputs .................................................................................. 112
Figure 0.D UPC relative speedup graph ..................................................................................... 113

Code Part 2.I Top level driver for task pool[31] ........................................................................... 25
Code Part 2.II A block distributed array example[24] ................................................................. 25
Code Part 2.IV Static, program managed load balance[31] .......................................................... 28
Code Part 2.V FSS Benchmark with collection of local arrays, implementing a global array[33] ...... 29
Code Part 4.I Chapel - Communication Phase 1 ......................................................................... 53
Code Part 4.II Chapel - Communication Phase 2 ....................................................................... 53
Code Part 4.III X10 - Communication Phase 2 ......................................................................... 54
Code Part 4.IV X10 - Communication Phase 2 ......................................................................... 55
Code Part 4.V UPC- Communication Phase 1 .......................................................................... 55
Code Part 4.VI UPC - Communication Phase 2 .......................................................................... 56
Code Part 4.VII Cofo rall loop example[55] ................................................................................ 57
Code Part 4.VIII The ateach construct[56] ................................................................................ 58
Code Part 4.IX Chapel - Types of iteration ................................................................................ 58
Table 4.A Language constructs with similar functionality in Chapel, X10 and UPC.......................... 60
Table 4.B X10 – Runtimes – Distributed Memory............................................................................. 65
Table 4.C X10 speedup table - Distributed Memory........................................................................... 66
Table 4.D UPC runtimes - Distributed Memory .................................................................................. 67
Table 4.E X10 runtime graph - Shared Memory ................................................................................ 69
Table 4.F X10 speedup graph - Shared Memory.................................................................................. 70
Table 4.G UPC runtime graph - Shared Memory ................................................................................ 71
Table 4.H UPC speedup graph - Shared Memory .............................................................................. 71
1 Introduction

1.1 Background

The limit of possibilities on traditional CPU computing has been reached, according to Professor of Engineering at Stanford University Bill Dally, and parallel processing is the next step towards software evolution. The emerging hardware architectures comprise of multicores, many-cores, clusters of multi-cores and even multi-core co-processors, like Intel’s Xeon Phi, or heterogeneous systems comprising of CPU’s and GPGPU’s. The challenge for the computing industry, is the development of complete and powerful languages, that will be adopted by the HPC programmers, overcoming the resistance to change.

Within this framework, the Defense Advanced Research Projects Agency of the United States (D.A.R.P.A) of America, started an initiative in 2000, called High Productivity Computing Systems (HPCS), aiming to promote productivity. The HPCS program has been a focused research on the creation of a new generation of high productivity languages to improve the performance of parallel programmers. The program explores different aspects, including computing systems, software tools, architectures and hardware components. The program has funded the development of three new languages - Chapel by Cray Inc, X10 by IBM Research and Fortress by Sun Microsystems.

One year earlier, in 1999, the Institute for Defence Analyses (IDA), Center for Computing Sciences, published a technical report, introducing Unified Parallel C (UPC, version 0.9). The coordinated efforts of universities, governmental funded organisations and HPC vendors resulted in the first UPC workshop in 2000, where version 1.0 was discussed. Some of the participants were: University of California, Berkeley (UCB), Sun Microsystems, Compaq, US Department of Defence and Cray Inc.

The project aims to perform a detailed assessment on programmability and parallel performance of the three developed languages. Programmability will address issues of language characteristics and pragmatics. We will review tool support, tutorial availability, publications and community support for each of the languages, as we consider these important for the uptake of a new programming language. Also, we will attempt a qualitative assessment on the ease of learning, based on the author’s programming background, representative of someone new to parallelism. On the issue of parallel performance we will develop the same application in the three languages and we will argue on performance results, on a comparative basis.

1.2 Aims & Objectives

The objectives of the project can be divided in two parts. In the first part, we aim to assess the three parallel languages, Chapel, X10 and UPC, in terms of features, current state and community activity. Also, we will make a brief review of the supporting tools and an assessment on the ease of
learning. In the second part, we intend to implement three versions of the N-body algorithm in these languages and compare the parallel performance and scalability of the produced programs.

The objectives can be divided in two categories: main and secondary.

- **Main objectives**
  
  - **Assess the usability and performance of state-of-the-art PGAS languages for parallel programming on commodity clusters.**

This assessment is our main requirement for this project. We intend to develop the same application (N-body all-pairs algorithm) in the three languages and compare them. Usability refers both to available parallel constructs that provide “easy” parallelism and also tutorials, development tools and community’s support (see 3 Small Scale Language Assessment). Performance refers to the quantitative comparison of runtimes and parallel speedup (see 4.3 Performance Comparison).

- Learning of the three languages

Learning of the three languages is the first step towards the assessment and will also indicate availability of learning materials and similarities to mainstream parallel and non-parallel languages (see 3.3 Ease of learning – Qualitative Assessment)

- Comparison and discussion on performance(in terms of speedup) and scalability (in terms of amount of code) for the implementations

Parallel performance comparison is a critical point for this assessment. We will argue on whether the novel constructs, implemented in the languages, differentiate the performance results or give easy solutions for parallelism (see 4.3 Performance Comparison).

- Comparison of the code’s quality and maintainability

The production of stable and maintainable code is a strong requirement in parallel programming. We will assess the quality of the produced code and the opportunities for clean interfaces that each of the languages offers (see 4.2.8 Code quality).

- **Secondary objectives**
  
  - **Discussion on the programming procedure of the three languages**
  - Discussion on the ease of installation and use of compilers and development tools

We aim to provide a review of these pragmatic issues, as they indicate the accessibility of the languages to new programmers (see 3.1 Tools & Compilers).

- Assessment of tutorial and documentation availability
The available documentation is a major issue in the learning of a new language. The existence of tutorials and examples, in addition to the official documentation of the language, give a clearer picture and facilitate the first stages of the learning procedure.

- Identification of the state and the current activity on the three languages

The status of the languages and the current activity of the community are indicative of the acceptance of the languages by HPC programmers and the ongoing work. (see 3.2 Current State).

1.3 Requirements Analysis & Methodology

1.3.1 Small Scale

I. Feature comparison

In order to assess and compare the features of the three languages we will refer to the languages specification reports and the relevant bibliography from published sources and code samples.

II. Current state

The current state of the languages will be reviewed in terms of new publications in scientific journals. Furthermore, mailing lists and discussion forums concerning the three languages will be monitored throughout the project. The rate of new publications and forum topics, will indicate the amount of work done on the languages and the dynamics of the user community.

III. Development tools

We, also, intend to make a brief review of the development tools provided for each language. The according web pages and forums will be used to locate the tools. Since the languages are not yet standardised, it is expected that development tools will be limited. Afterwards, we will set up the environment and install the tools. The amount of existing user guides and installation manuals will also give information on the current state of the languages.

IV. Qualitative assessment

The assessment of the learning procedure will be qualitative. As the process of learning a language depends on the programmer’s background, the approach we will adopt, is based on the similarity of the concepts introduced in the three languages to other mainstream languages, such as Java and C/C++. Also, we will give an approximation of the time invested to learn and assimilate the main features of each language and the time needed to implement preliminary trivial applications.
1.3.2 Large Scale

I. Presentation of algorithms

In the second part of the project we will make a brief presentation of the different algorithms proposed for the N-body problem, based on the related bibliography. We will then choose the most suitable for parallel development and we will produce the first applications.

II. Application development and performance

As far as the performance of the application is concerned, we will test and compare the runtimes on the Beowulf cluster, using statistical quantitative methods and producing corresponding graphs. The same approach will be followed in order to assess the programs’ scalability; we will perform testing with different input sizes.

III. Code quality

Finally, we will argue on the code quality. For this discussion, we can use measurements on the number of lines of each program or check the existence of generic types in the code. Maintainability will be addressed in terms of independence from the underlying hardware features. We could also discuss the use of independent modules and called functions in the programs.

1.4 Professional, Legal, Ethical and Social Issues

The tools, compilers and libraries used in this work are part of Open Source projects and they are freely available. No licence or special permission is required to use them. The N-body benchmark code in C and Java, which we have used as guideline, is contributed by Mark C. Lewis.

Since, no people, questionnaires or user testing is involved, there are no particular ethical issues related to this project. Subjective assessment of issues, such as ease-of-use and programmability, reflect the authors own experience

Performance evaluation will be performed on the Beowulf cluster, available to the students of the Department of Mathematical and Computer Sciences.
1.5 Structure

The content of the project is structured as following:

Chapter 1: Introduction – Describes the background, highlighting the reason that led to this research topic and sets the objectives of the project. Also it sees the detailed list of aspects we have investigated and compared.

Chapter 2: Literature Review – Reflects on the current state in the field of parallel computing, introduces the three languages under comparison and their characteristics and specifies the application we used for the assessment.

Chapter 3: Small Scale Language Assessment – Sees the aspects of tool support and current status of the assessed languages and comments on the ease of learning.

Chapter 4: Large Scale Language Assessment – Consists of the second part of the core assessment with the detailed review of the produced code, comparing between the three languages, and the performance evaluation of the programs.

Chapter 5: Conclusion – Covers the summary and evaluation of the project in terms of what has been achieved, and points to future work.

Appendices: Listing of the final versions of the source code and additional material (speedup graphs etc).
2 Literature Review

2.1 The evolution of machines

The continuous development of computers and computational technology has brought the programming world into the era of high speeds and multi-core machines. Today, a conventional home machine is at least dual or quad-core and can achieve around 15 gigaflops of computational power. The sequential performance (in terms of clock frequency) has stalled, while the number of cores per chip is increasing rapidly, as a result a laptop today is equivalent to the machines used in supercomputing ten years ago. In order to achieve the best performance from the growing number of high performance machines, the development of new, more sophisticated parallel programming languages becomes a necessity. From a hardware’s aspect the goal is doubling the number of cores on a chip per year, as Berkeley’s scientists claim [1].

2.2 The need for parallelism

Single-threaded processor performance rate has remained immutable from 2005 (between 0 - 5 % per year) [2](Figure 2.A Moore’s Law According to the graph we could continue increasing transistors, but Moore’s Law makes no prediction on performance. Power cannot be increased or chips will melt. Processor frequency cannot scale further.). This is mainly attributed to the stagnant clock speeds, both due to unresolved issues of power consumption and chip scalability. In order to speed up the applications, the focus is now shifted to parallel computing. Languages, tools and API’s for parallel programming are improving rapidly. In 2005, the president of Intel, Paul Otellini, stated: “We are dedicating all of our future product development to multi-core designs. We believe this is a key inflection point for the industry”, describing Intel’s future directions.

![Microprocessor Transistor Counts 1971-2011 & Moore’s Law](image)

Figure 2.A Moore’s Law: According to the graph we could continue increasing transistors, but Moore’s Law makes no prediction on performance. Power cannot be increased or chips will melt. Processor frequency cannot scale further.
2.3 Parallel Architectures

2.3.1 Memory models

2.3.1.1 Shared memory model

2.3.1.1.1 The Hardware Perspective

A shared memory architecture consists of a number of independent processors, a set of memory modules and a network that handles their interconnection. Hosts gain access to remote memory through message requests (message passing). A memory controller is used to administrate requests by forwarding the requests when the memory is idle or by sending waiting signals when the memory is busy. An overview of the shared memory model is shown in the figure.

Based on the interconnection network, we may distinguish three categories of shared memory architectures. Uniform Memory Access (UMA), in which all processors have balanced access to shared memory. Non Uniform Memory Access (NUMA) where a part of the shared memory is attached to each processor. The access time here depends on the distance to the processor. An example architecture is Cray T3E.

An alternative on shared memory architectures, developed during the 1990’s, is the Cache-Only Memory Architecture (COMA), where each processor has part of the memory, but the memory is composed exclusively by caches. A cache directory helps migrating the data to the requesting processors.

In the shared memory model, the number of cores is scaling the memory architecture and the design of a cache coherent system becomes complex. The ongoing research on the elimination of the complexity, proposes an approach based on non-cache-coherency, which avoids sharing of data and relies on message passing [3]. An example of non-coherent-caches is the Intel SCC (Single-chip Cloud Computer), announced in 2009. Another approach to this matter, proposed by the Compiler Microarchitecture Lab of Arizona State University, includes a virtual shared memory, implemented by a library which updates the caches.
2.3.1.1.2 The Software Perspective

In this model, the memory is mapped into the address space of the processes that are sharing the memory region. There is no passing of data from one process to the other so the kernel is not involved. Due to the fact that all processes have simultaneous access to the memory, there is the need for some form of synchronization when storing or retrieving data from a shared memory chunk. From a programmer’s view, synchronization techniques such as mutexes, condition variables, read–write locks, record locks, and semaphores are regularly used [4]. OpenMP is the most popular API for writing parallel programs, for this type of memory.

The main advantage of a shared memory system is the ease of programming, since interprocessor coordination and synchronization is managed via the global memory. The main challenges of shared memory systems are performance and scalability. Performance is affected by the contention of the interconnection network and the memory consistency problems between global memory and caches. Therefore, shared memory systems do not scale efficiently.

2.3.1.2 Distributed memory model

2.3.1.2.1 The Hardware Perspective

A distributed memory architecture consists of individual processors, each with its own memory and I/O mechanism. The processors use an interface to interconnect in a global network. It is also possible that a small number of processors function as an individual node in the system, connected internally by a bus. The address space here is formed by a number of disjoint address spaces. The same physical address can refer to different locations (different processors’ private memory), thus precluding explicit access from a remote processor. Clusters use this type of memory architecture [5]. Figure 2.C gives an overview of the distributed memory model.

2.3.1.2.2 The Software Perspective

In the distributed memory model, processes use their local memory during computation. A set of processes resides in the same physical machine or in a networked environment. The need of data exchange is fulfilled by message passing. Processes cooperate to form tuples of send/receive actions. Implementations of message passing usually consist of libraries of subroutines. The programmer is responsible of embedding subroutine calls in the source code [6]. MPI (Message Passing Interface) is the standard for performing message passing on distributed memory architectures.
A distributed memory system can achieve high memory bandwidth and reduction of latency, provided that most accesses are to the local memory. The key disadvantage is that programming for such an architecture requires more effort, in order to coordinate the exchange of data and make use of the high bandwidth available.

2.3.2 Programming models

In an effort to disengage the design of parallel algorithms from the characteristics of the underlying technology, additional programming models have been developed. The design introduced in these models can be distinguished based on the amount of hardware details they intend to hide from the programming language. Thus, they can be separated to implicit (as GpH), explicit (as MPI) or semi-explicit. The novel languages, discussed in this project, are developed to offer implicit parallelism, mainly in the form of default parallel constructs.

2.3.2.1 Hybrid Programming Model

A hybrid system combines a set of shared memory architectures, networked in a distributed manner. Processes can access data both in the local shared memory and fetch data from remote memory via messages. The motivation for the Hybrid Programming model is the achievement of higher performance, building on a message passing base code.

The main advantage of this model is flexibility as it combines benefits of both architectures. The disadvantages are detected in the implementation difficulties that occur. As data updates are handled both by the user and the system, synchronization is essential. Furthermore, work division must be assigned efficiently to user and compiler. Finally, operations such as reductions and initializations are prone to exceeds memory bandwidth [7].

The hybrid model is a current trend which seems to be prevailing in the future, mostly due to its flexibility and the combination of advantages of both architectures [8]. A commonly used example of the hybrid model combines the MPI model, for communication between processes over the network, and the threads model (OpenMP), to execute the computational parts locally, in every node. This example ranks among the explicit models, since the programmer is required to tune the communication and distribute the workload among processes [6].

2.3.2.2 Task/Channel Programming Model

The Task/Channel model is a proposed abstraction for parallel algorithms, addressing concurrency and locality. Tasks consist of the sequential parts of code executed in each parallel node, along with their local memory and their I/O ports. Apart from the computation tasks perform, they also execute four main operations. These operations are: Sending of messages, receipt of messages, creation of other tasks (children) and termination when the computation is complete.

Tasks are a concept similar to objects from the Object Oriented Programming, since they both consist of data and code operating on the data. Tasks can be mapped to the underlying
architecture, meaning the number of available processors. Channels implement the communication between tasks and are, in essence, FIFO ordered message queues. Channels can be dynamically created and deleted to match the tasks’ grid structure and interconnections [8].

2.3.2.3 Skeletons

Skeletons are high-order templates for parallel implementation, which capture patterns of common applications. Skeletons provide structure, but lack detail. Based on extended research, they are proven to give predictable performance [9]. The use of a skeleton allows the programmer to take advantage of the parallel behaviour, simply by adapting the sequential implementation to the requirements, such as process placement and interconnectivity, instructed in the definition of each skeleton model.

MapReduce is a computing model developed and used by Google for efficient distributed computation over large data sets, on clusters of computers [10][11]. As the name suggests, it is based on two core functions, ‘map’ and ‘reduce’, inspired by functional programming. This idea of parallelism offers recovering possibilities from partial failure of servers during the operation. In case, a node fails, the operation can be rescheduled provided that the input data is still available. With the use of MapReduce on a large server farm, it is feasible to achieve sorting of a petabyte of data in only a few hours [10] [11].

2.3.2.4 Partitioned Global Address Space (PGAS) Programming Model

The Partitioned Global Address programming model provides a unified address space, hiding the details of the underlying memory architecture. Data is logically partitioned between local and global memory and they are directly accessible by any process. This kind of data affinity addresses memory latency issues [10]. Synchronization is achieved by optimised one-way communication between processes and by the compiler’s intervention [12].

The PGAS programming model can execute on shared memory using a two-level hierarchy of one or more processes, each with one or more threads. It can also execute on a distributed memory system, mapping one or more processes to each node. Finally, using a three-tier hierarchy of nodes, processes and threads, PGAS can adapt to a hybrid execution environment [10].

PGAS is developed to allow partitioning, data locality and performance, to co-exist with the minimalism of referencing in shared memory, thus integrating the beneficial features of the two
worlds [12]. The model employs the concept of implicit parallelism, alleviating the burden of defining parallelism from the programmer.

The languages discussed in this project represent part of the PGAS family of languages, along with Titanium by University of California, Berkeley and Co-Array Fortran as an extension to Fortran 95, decided by the ISO Fortran Committee.

2.3.3 Widely Used Parallel Languages/ Libraries

2.3.3.1 C+MPI (Message Passing Interface)

The Message passing Interface is a standard library specification for implementation of explicit parallelisation with message passing. In MPI, one or more processes communicate to perform a computation. The processes call library routines to send and receive messages [66]. It is possible for every processor to execute different code, thus making MPI applicable to the MIMD and the SPMD model. In most cases, each process will execute on a different processor, will have a unique id and will use point-to-point communication to exchange data.

MPI’s distinguishing feature is the communicator, which identifies the context and the processes related to each computation. The message passing interface targets distributed memory [13], but also shared memory architectures [14], although the latter is considered harder to implement [15].

2.3.3.2 C+ OpenMP (OpenMultiProcessing)

OpenMP is an API for writing multithreaded applications, introducing a higher level of abstraction compared to MPI. It consists of a set of compiler directives, executed by the preprocessor, library routines, and environment variables, used to specify shared memory parallelism. OpenMP has bindings to Fortran and C/C++ [16] and it addresses shared memory and hybrid programming.

Parallelism in OpenMP is introduced with the compiler directives, without any changes to the binding language. A number of threads is spawned to parallelise the high consuming parts of the sequential code[17], thus addressing task parallelism.

It can deliver scalability for applications, using large domain decomposition blocks and dynamic load balance. On the other hand, OpenMP does not render satisfactory performance for data parallel algorithms, it cannot address algorithms build out of components and it achieves 2-level modularity at most [17][16].
### 2.3.3.3 Glasgow Parallel Haskell (GpH)

GpH is an extended parallel version of Haskell, including primitive concepts to facilitate parallel programming. It offers a variety of built-in types and abstraction mechanisms, thus it can address parallel programming purposes and provide architecture-independent parallelisation [18][19].

Parallelism is introduced in the form of evaluation strategies, which are high-order functions determining the parallelism and the order and depth of the evaluation. Due to the use of strategies, the original language (Haskell) is allowed to remain unchanged [20] [21].

GpH addresses distributed memory architectures, implementing a virtual shared heap, while many different approaches have been proposed on its implementation. One interesting approach is that of eliminating any locks or memory barriers from the evaluation, to improve performance [22].

### 2.4 Chapel, X10 & UPC

#### 2.4.1 Development motivation

The motivation for the HPCS program came in the early 2000’s, as the program was meant to benefit different scientific areas and national security issues, such as Operational Weather and Ocean Forecasting, Signals Intelligence, Intelligence, surveillance, and reconnaissance, Weapons design, Biotechnology and many more [23]. These types of applications, could only be funded by the government, since they were not targeted to the market, so they could not be assigned to manufacturers.

The motivation for the development of UPC occurred as an attempt to combine the successful parallel characteristics and knowledge gained from previous extensions to C 99, like AC, Split-C and Parallel C Preprocessor and in the same time refine them.

Both universities and industries were involved, in the three phases of the HPCS program and in the development of UPC.

#### 2.4.2 Requested features

The predominant design requirements for modern programming languages, also reflected in DARPA’s HPCS Program [2], can be summarized to the following:

- **Performance**: Targeting the improvement in computational power by 10 to 40 times, over the current performance rates [2].
- **Programmability**: Targeting to decrease development time and maintenance to 1/10 [2].
- Portability: The result products need to be independent of software or hardware architectures [2].
- Robustness (Reliability): Fault tolerance in hardware and software defects [2].
- Heterogeneity: Ability to address a range of emerging architectures, as co-processors and GPGPU’s and combinations of the above, with equivalent performance results.
- Deterministic parallelism: The property of receiving the same result for the same input in each execution. This is a weaker requirement, though capable of providing sound formal basis for the produced programs.

The performance requirement refers to the need for more powerful machines and predates the development of multicores. In the following discussion of the languages’ characteristics, we will identify a number of design directives, which illustrate the designers’ tendency to address parallel computation tasks. Furthermore, the portability goal refers to the adaptability of the produced programs to different architectures, implying performance portability irrespectively of the underlying software and hardware.

2.4.3 The PGAS family of languages

The languages representing the Partitioned Global Address Space model share the general aim of increasing programmers productivity by simplifying parallel programming [24]. The PGAS model exposes both data and thread locality in order to improve programmers performance (development) and parallel performance of programs (execution) [25]. The three languages assessed in this project (Chapel, X10 and UPC) are part of the PGAS family, along with Titanium developed by Berkeley, University of California, and Co-Array Fortran, which is an extension to Fortran 95 decided by the ISO Fortran Committee.

2.4.4 Language characteristics

2.4.4.1 Chapel (based on specification v0.93) [26]

Chapel was developed by Cray, as a proposal to DARPA’s high performance challenge. It is a language designed from first principles rather than extending existing languages, but at the same time, a derivative of a number of parallel and distributed languages. It borrows ideas, such as the global view programming model, inherited from High Performance Fortran and ZPL. Chapel also borrows sequential object oriented features, as classes, objects and records, from C#, C++, Java and Ada. The main goal in Chapel’s development is to increase productivity of parallel programming and the main focus is on programmability.

Chapel’s design is based on the principles of general parallel and locality-aware programming, to support parallel performance by high-level abstractions. Furthermore, it relies on object oriented programming, taking advantage of the clean implementations provided from the encapsulation of data and functions in the same software component. It, also, encompasses generic
programming features, facilitating code reuse irrespectively of the variables’ types and thus addressing mainstream programming and scripting languages. It is a statically typed-safe language, meaning that the types of the variables, and thus the possible set of values, are known during compilation. Interoperability is significantly enhanced allowing the writing of programs with both Chapel and non-Chapel components and the call of external and Chapel functions in the same program.

Parallelism in Chapel is introduced in the form of task, data and nested parallelism. Each computation is assigned to a collection of threads, in an implicitly multithreading manner. The management of low level threads is left to the compiler and the runtime system [27].

2.4.4.1.1 Sequential Model

Chapel borrows the concept of objects from OOP, allowing data and operations to coexist in a single component. This is particularly useful for code reuse and the development of clean interfaces. Object oriented programming style is achieved through library functions, so it is not a restrictive factor for other programming styles. Generic coding is also supported, allowing the development of constructs of many types, without explicit definition for each of them. Generics enhance flexibility and reduce runtime overheads.

Apart from classes, it supports records which are similar constructs, but assignments, argument passing and return values are performed by copying (value semantics, similarly to C#). Records differentiate themselves from classes in terms of storage allocation, declaration, assignment, argument passing, inheritance, overriding and default comparison operators.

Unions are structures similar to records, but allowing only one data field in every execution point. Unions are initialised in an unset state, empty of data. Ranges are first-class space representations of sequences of integer indices, supporting iteration over them.

Chapel’s support for multiple inheritance, namely allowing a class to declare base classes from which it inherits functions, is currently an open issue. In any case, if multiple inheritance is finally used, then only one of the base classes will be allowed to contain fields while the methods of the rest base classes will be possibly overridden by the inheriting class, similarly to C#’s interfaces. This constraint forces a tree-structured class hierarchy.

Finally, Chapel provides a rich set of libraries in the form of automatic and optional modules. The automatic modules are Math, Base and Type, and examples of optional modules are BitOpts, Search, Sort and Time.

2.4.4.1.2 Control Structure

Chapel supports the Global View Model to express data and flow control concepts, raising an abstraction with respect to commonly used parallel languages. Parallelism is more general than the SPMD model, and it is introduced via multithreading. Low level thread management is performed in
the compiler, aided by the runtime system. This allows Chapel to target multiple parallel architectures, writing architecture-neutral programs, with ensured performance portability.

**Task parallelism** is supported by established constructs that introduce concurrency (begin, cobegin, coforall), synchronize tasks (synchronization variables), control or suppress parallelism (synchronization and serial statements) and atomic transactions (atomic statements). The feature of atomic statements implies the existence of a transactional memory scheme, implemented in the compiler and the libraries [28]. In contrast to what is stated in the specification, atomic statements are not supported in version 1.7.0. **Data parallelism** is introduced via forall statements and expressions, promotion, reductions, scans and configuration constants.

**Iterators** are another special concept in Chapel. Iterators are functions that yield values, as in C#, consecutively or in parallel. Parallel iterators are used in explicit forall/coforall statements and they are predefined for ranges, domains and arrays. The use of a coforall statement over a range is shown in Code Part 2.1 Top level driver for task pool[31](lines 4-7) . Input/output facilities include file and channel types, stdin, stdout, stderr constants and functions as write/read and open/close. The concept of iterators is introduced in Chapel to factor the data structure traversal away from the computation and avoid the complex nested loops which impose overheads to the algorithm [29].

There exists a predefined set of operations on domains, including operations such as: iteration, using **for**, **forall** and **coforall** loops and passing a domain as argument.

### 2.4.4.1.3 Communication and Placement Control

**Domains** are a major construct of Chapel. They represent ordered sets of Cartesian indices [4], specifying iteration spaces, aggregate operations, as slicing, and defining the size and shape of arrays. This feature is highly desirable, as it is often useful in parallel numerical computation. Iteration spaces can be single or multi-dimensional and a single domain's indices can be distributed across multiple locales.

**Locales** are referred to, as “portions of the targeted parallel architecture with processing and storage capabilities”. Locales serve the purpose of allowing the tasks to have uniform access to the data stored in the same locale, introducing an abstraction over a processor, as CPU – memory pair. As a result, access to remote locales’ data increases memory latency. A locale is a primitive type, to which data and tasks can be associated and equality operations can be applied. The **on** statement is used to define the local on which locale tasks are executed or data are stored.

Domains and arrays are mapped on **domain maps** that specify their implementation. Domain maps define how indices and array elements are mapped to locales and how operations on them are performed. Domain maps are encountered in most PGAS languages and they can be either layouts or distributions, describing domains and arrays on the same locale or partitioned across multiple locales. Commonly used distributions, such as **block** and **cyclic**, are predefined to reduce the performance cost, but user defined distributions can be developed to target specific applications [30].
Locality is introduced by the option to specify where to place data and computation on the physical machine. Although, this feature contradicts with shared memory and SPMD programming model, it solves valuable control issues in distributed memory architectures.

Chapel’s memory consistency is only guaranteed for race-free programs, addressing less strict security requirements, compared to Java’s memory consistency.

2.4.4.1.4 Code examples

```chapel
config const numConsumers = 10, poolSize = 10;
    const t = taskpool(poolSize);
    cobegin {
        coforall ons in 1..numConsumers do
            consumer();
            producer();
    }

Code Part 2.1 Top level driver for task pool[31]
```

The above example uses the cobegin statement to concurrently execute the commands in its body. The coforall statement uses a parallel iterator to iterate over the given range. In the concurrent part, two functions are called.

```chapel
use BlockDist;
config const n = 8;
const Space = (1..n, 1..n);
const BlockSpace = Space dmapped
    Block(boundingBox=Space);
var BA: [BlockSpace] int;
forall ba in BA do
    ba = here.id;
writeln("Block Array Index Map");
writeln(BA);
```

```chapel
var MyLocaleView = {0..#numLocales, 1..1};
var MyLocales: [MyLocaleView]
    locale = reshape(Locales, MyLocaleView);
config BlockSpace2 = Space dmapped
    Block(boundingBox=Space,
        targetLocales=MyLocales);
var BA2: [BlockSpace2] int;
forall ba in BA2 do
    ba = here.id;
writeln("Block Array Index Map");
writeln(BA2);
```

```chapel
var MyLocaleView = {0..#numLocales, 1..1};
var MyLocales: [MyLocaleView]
    locale = reshape(Locales, MyLocaleView);
config BlockSpace2 = Space dmapped
    Block(boundingBox=Space,
        targetLocales=MyLocales);
var BA2: [BlockSpace2] int;
forall ba in BA2 do
    ba = here.id;
writeln("Block Array Index Map");
writeln(BA2);
```
Code Part 2.II A block distributed array example[24]

```plaintext
use CyclicDist;
const CyclicSpace = Space dmapped Cyclic(startIdx=Space.low);
var CA: [CyclicSpace] int;
forall ca in CA do
  ca = here.id;
writeln("Cyclic Array Index Map");
writeln(CA);
writeln();
```

Cyclic Array Index Map
0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3
4 5 4 5 4 5 4 5
0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3
4 5 4 5 4 5 4 5
0 1 0 1 0 1 0 1
2 3 2 3 2 3 2 3

Code Part 2.III A cyclicly distributed array example[24]

The above code snippets show two of the standard distributions provided in Chapel, Block and Cyclic. In Code Part 2.II A block distributed array example[24], the Block distribution uses the Space range as bounding box and in the second case the Locale’s array is specified producing different index mapping. The cyclic mapping result is demonstrated in the figure of Code Part 2.III A cyclicly distributed array example[24].

2.4.4.2 X10 (based on specification v2.3)[32]

X10 is a class based object oriented language developed by IBM. Its name suggests IBM’s intention for ten times boost in productivity. X10’s design principles are *asynchrony, locality, atomicity* and *order*. By asynchrony we mean the Asynchronous PGAS model introduced, which permits the execution of both local and remote asynchronous tasks. Locality refers to the abstraction over a computational context, used by X10. Atomicity, refers to the atomic way each task accesses and modifies data in a location, while order implies means provided for synchronization of the different running tasks.

The design goals are “high performance and high productivity computing on high-end computers” as stated in the specification of the language. X10 is largely based on object oriented languages, differentiating only to support its design goals. X10’s targeted application areas are numerical computations and commercial server workloads. It is intended to have clear semantics, support concurrent programming and be accessible to object oriented programmers. X10 is part of the PGAS family and extends the model by introducing Asynchronous PGAS (APGAS).

2.4.4.2.1 Sequential Model
The object oriented ideas which it assimilates have the advantage of proven flexibility and facilitate the learning and use of the language. The sequential core of X10 is container-based, similar to Java, C++ and Scala. It provides classes organized in a single inheritance tree, thus having a single parent, but implementing multiple interfaces. It also provides structs which are similar to classes but less powerful. Structs cannot inherit methods, but they are cheaper in terms of space requirements. Structs can be inlined and are immutable during execution. There are no primitive classes in X10, although the standard library provides Byte, Boolean, Short, Char, Int, Long, Float, Double, Complex and String. Structs are, also, user defined and interfaces are collections of abstract methods.

Functions are first class data, used to allow operations on values. Functions can be stored in lists and passed as arguments and they are extensively used in X10. Constrained types are immutable fields bound to an object or a struct. They look like type names, followed by Boolean expressions. Constrained types can be returned as function and constructor values. Finally generic types of objects are supported.

The sequential core of the language is very similar to C. “As” operations allow explicit conversions of variable types, and implicit coercion is also allowed. Common control flow constructs, such as if, while, for loops, switch and try-catch are used with their known functionality.

2.4.4.2.2 Control Structure

X10 is designed to address scalability, productivity and support interoperability. The focus on local data and the limited ability of events to delay other events in a remote place, enhance scalability opportunities. Furthermore, unconditional atomic blocks are non-blocking and the data flow synchronization allows coordination between reader and writer of the same data, thus the use of locks and barriers is reduced.

Productivity can be split in two aspects. Safety features such as, type safety, memory safety and pointer safety, are essential. Integration with other existing constructs is also relevant to programmers’ productivity. By type safety, it is meant that, a location only contains values matching its declared type and method calls can be statically or dynamically checked to comply with type declarations. The basic principle, supporting memory safety, is that an object can only access memory within its representation or via objects it has references to. Every location is initialized at runtime, ensuring that a read operation occurs after a write operation (i.e. there exists data to be read). Memory safety, also, ensures place safety, since all local operations are actually executed locally and all local data are actually stored locally. Locks and unconditional atomic blocks are guaranteed not to end up in deadlocks. The second feature of productivity is integration. X10 is able to support a large number of existing libraries and tools, thus facilitating the programming procedure. Interoperability is also a decisive aspect of productivity. In X10’s case, programs can use Java and C (C++) components.

Objects are small sets of fields with a distinct name and can be distinguished based on the place they reside. Scalar objects which are located in a single place throughout lifetime and aggregate objects which may be distributed across several places with an unchangeable distribution during execution. Objects of all types are deleted from the memory with the help of the garbage collector, when there is no reference to them. The programmer cannot explicitly release memory.
As far as global address space use is concerned, X10 allows an activity to reference objects at remote places, but can only access data in the current place. Atomic operations are used to update local items and shift operations for reading and modifying objects on remote places. The concept of atomic blocks allows an activity to execute atomically, without interruptions from other activities. X10 also allows conditional atomic blocks, which is often useful but rarely available in parallel languages. Finally, asynchronous activities can be started without stalling the program till their completion. An example of an asynchronous tasks is given in Code Part 2.IV Static, program managed load balance[31], below(line 4).

The idea of clocks in X10, represents a computation that occurs in phases. An activity can be assigned zero or multiple clocks and operations such as waiting on a clock or starting new clocks are available. Clocks contrast to MPI’s barriers, as an activity can have many of them, yet deadlock free computation is guaranteed.

2.4.4.2.3 Communication and Placement Control

The two main concepts that introduce concurrency are places and activities. As X10 is intended to run on a large variety of processors, from uni-processors to large clusters, places are introduced as an abstract computational context with local synchronous view of data. Places represent data-coherent elements [31] supporting a finite number of hardware threads and an amount of uniformly accessed shared memory. An example or a cyclic-ordered set of places is shown in Code part 3 (lines 1 and 7). Activities, on the other hand, are executions of lightweight threads on data. Activities residing in the same place use atomic statements to access the local memory and weaker ordering semantics are provided for operations on distributed arrays [31].

Finally, there exists a special type of arrays, distributed arrays which may be distributed across multiple places and support collective operations, similarly to distributed objects in PGAS languages. A distributed array (and object) provides a distribution to inform about which elements can be found in which places. An X10 programmer is forced to explicitly deal with distribution following the belief that locality aspects cannot be hidden to the programmer of parallel programs. Regions are a high abstraction over arrays, shadowing the shape and dimensions of the underlying array.

2.4.4.2.4 Code examples

```x10
place placeNo = place.FIRST_PLACE;
finish for (point [iat]: [1:atom])
  for (point [jat, kat] : [1:iat, 1:iat])
    for (point [lat] : [1:(kat==iat?jat:kat)])
      async (placeNo)
        buildjk_atom4 (new blockIndices(...));
      placeNo = placeNo.next();
    }
  }
}

Code Part 2.IV Static, program managed load balance[31]
```

In the above code part, we see a cyclic ordering of places. Starting in the first place, each iteration launches a new asynchronous task in every place. Then the place is substituted by the next
place. The finish statement, forces the program to wait for the completion of all the asynchronous tasks. BlockIndices is a class that specifies the work to be performed by the tasks.

```java
static MEG = 1024 * 1024;
static alpha = 3.0 D;
static NUM_TIMES = 10;
static DEFAULT_SIZE = MEG / 8;
static NUM_PLACES = Place.MAX_PLACES;

public static def main(args:Array[String] (1)) {
  val verified = new Cell[Boolean] (true);
  val times = GlobalRef[Array[double] (1)] (new Array[double] (NUM_TIMES));
  val N0 = args.size > 0 ? int.parse(args(0)) : DEFAULT_SIZE;
  val N = N0 * NUM_PLACES;
  val localSize = N0;
  Console.OUT.println("localSize=" + localSize);

  finish {
    for (p in 0...(NUM_PLACES - 1)) {
      async at(Place.place(p)) {
        val a = new Array[double] (localSize);
        val b = new Array[double] (localSize);
        val c = new Array[double] (localSize);
        for (i in 0...(localSize - 1)) {
          b(i) = 1.5 * (p * localSize + i);
          c(i) = 2.5 * (p * localSize + i);
        }
        for (j in 0...(NUM_TIMES - 1)) {
          if (p == 0) {
            val t = times as GlobalRef [Array[double] (1)] {self.home== here};
            t()(j) = now();
          }
          for (var i:int = 0; i < localSize; i++)
            a(i) = b(i) + alpha * c(i);
          if (p == 0) {
            val t = times as GlobalRef[Array[double] (1)]
            {self.home == here};
            t()(j) += now();
          }
        }
        // verification
        for (var i:int = 0; i < localSize; i++)
          if (a(i) != b(i) + alpha * c(i))
            verified.set(false);
      }
    }
  }
}
```

Code Part 2. V FSS Benchmark with collection of local arrays, implementing a global array[33]

The above code uses a global array (GlobalRef) and performs the main computation inside a synchronised parallel for on the available Places, but asynchronously on each of them. All places have access to the global array.
2.4.4.3  UPC (based on specification v1.2)

2.4.4.3.1  Sequential Model

Unified Parallel C is a parallel version of the C standard. The sequential core of C is the base on which UPC was developed, and all its characteristics are also inherited by UPC. Functions are the predominant construct and structs replace objects. Any legal C program is also considered legal in UPC [34].

2.4.4.3.2  Control Structure

UPC follows the Single Program Multiple Data (SPMD) execution model, also used in message passing. The existence of threads and their explicit handling is the main feature of the language. The program starts on one thread (Thread 0) and produces more threads, through the declaration of shared objects and the use of parallel constructs. UPC threads provide an abstraction and they can be implemented as OS processes or as user/kernel level threads. The only requirement is that a parallel underlying architecture running a UPC program must provide at least one UPC thread per available processor. Shared objects here, refer to distributed arrays or data, which are available to all executing threads, and not to the familiar construct from Object Oriented Programming. The shared type qualifier is used to mark data which are referred to with the same address on each thread.

UPC introduces the ideas of affinity and access. Affinity refers to the logical relation between an object and the thread which owns it. Each of the elements of a distributed structure (array) has affinity to exactly one thread, while scalar data have affinity to Thread 0. Access is an action performed during execution, and refers to the reading or modification of a value by a thread.

Accesses are separated to shared and local. While there is no syntactic difference in the language, accesses are distinguished by the use of the according qualifier (shared qualifier for a shared access) of the pointer. Furthermore, all shared accesses are either strict or relaxed. A strict access occurs in program order to all threads and guarantees sequential consistency (Lamport). A relaxed access does not appear in program order except if the first one is a “write” operation and the other is a “read” performed by the same thread. Access consistency can be specified on per variable or per statement basis.

In this distributed environment, synchronisation is essential to preserve memory consistency. UPC specifies barriers, locks and signals to achieve synchronisation. Some examples are: upc_fence, upc_barrier, upc_notify, upc_wait, upc_lock and upc_unlock. All these primitives are explicitly handled by the programmer.

Finally, a UPC program terminates when all of its threads have finished executing or when a call to the function upc_global_exit() is made.
2.4.4.3 Communication and Placement Control

The placement of shared objects on the available number of threads is managed according to the type of data. Arrays are handled using layouts. UPC offers four types of layouts: cyclic, blocked, indefinite and fixed block size. Cyclic and blocked layouts have the usual functionality. The indefinite layout, maps all elements to one thread, while the fixed block size layout allows the programmer to specify the block size allocated to each thread. In contrast, scalar data relate (have affinity) to Thread zero.

The objects declared with one of the above distributions are called shared objects. All threads may access shared objects. In contrast, there exist private objects, which are declared and accessed only by one thread (the owner thread).

Another feature introduced in UPC is the distinction of C pointers. In UPC there are pointers-to-shared and pointers-to-local. A thread can produce pointers-to-shared for all the shared objects in the program and pointers-to-local for its own objects (objects with affinity to this thread). In the same time, the thread can access the “local” object with a pointer-to-shared (although casting is recommended), and if the shared accesses in the program are relaxed, a thread can point to a shared object using a pointer-to-local. The table below summarizes the allowed types of pointers and their declaration.

<table>
<thead>
<tr>
<th>Pointer points to:</th>
<th>Local</th>
<th>Shared</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private p1</td>
<td>p1</td>
<td>p2</td>
</tr>
<tr>
<td>Shared p3</td>
<td>p3</td>
<td>p4</td>
</tr>
</tbody>
</table>

Finally, collectives are used to evaluate operations across all threads of the program. A collective can be single valued, for example using an operand with the same value on each thread (for example an addition of all elements of a shared array of integers). Also a collective can specify a phase on each pointer to indicate the element’s offset within an affinity block.

2.4.4.3.4 Code examples

```c
shared[*] int v1[N], v2[N], sum[N]; // blocked layout
```
void main()
{
    int i;
    for (int i = 0; i < N; i++) {
        v1[i] = 1;
        v2[i] = 3;
    }
    upc forall (i = 0; i < N; i++) & sum[i])
    {
        sum[i] = v1[i] + v2[i];
    }
    upc forall (i = 0; i < N; i++)
    {
        printf("%d", sum[i]);
    }
}

Code part VI Vector addition using blocked shared array [34]

2.4.4.4 Feature Comparison

From the previous discussion of the languages’ features, there have been identified some high level similarities and differences.

Conceptually, Chapel and X10, rely on the object oriented and the generic programming model, to structure the base language and make extensive use of default libraries. The control of parallelism is, also, employed in a similar way. They trust the high level coordination to the programmer and handle parallel optimisations in the compiler and the runtime system [31]. UPC on the other hand complies with C’s procedural programming style, extending it with the concept of shared objects. The explicit use of threads, instead of another form of locality abstraction, offers the potential for low level coordination.

All three languages follow the partitioned global address space memory model, and especially Chapel and X10 introduce asynchrony to the global memory view (APGAS). Chapel also, uses constructs that imply transactional memory [28].

Abstraction over location is another common feature between Chapel and X10, although the details are implemented differently in each of the languages. Chapel names these abstractions locales and X10 uses places. On the other hand, UPC directly handles threads, although some level of abstraction is provided by the distributions (layouts) of shared objects.

Chapel and X10 support the concepts of inheritance and polymorphism. Inheritance can be either single or multiple.

From a more technical aspect, Chapel and X10 use one thread (thread 0) in the beginning of the control flow and produce more parallelism using constructs. UPC uses a different execution model. All the available threads execute an instance of the program independently in SPMD style.

In Chapel and X10, functions are first order constructs and they all employ type safety, to avoid erroneous situations due to differences between variables’ types. UPC programs arrange all the executable code in functions, and following C, UPC is type safe in limited context. For example, casting between different types of pointers is handled by the programmer.
Finally, they all provide support for foreign function interfaces, calling functions or methods on objects, which are written in other languages or libraries. A very interesting work, investigating the benefits of combining UPC with MPI functions is presented in [35].

2.5 Designing Parallel Algorithms

The following design approach is aimed to introduce concurrency early on in the algorithm and target machine specific issues later in the procedure. The design of the algorithm is divided into four stages: Partition, Communication, Agglomeration and Mapping. The first two stages focus on the achievement of concurrency and scalability and the following two focus on locality and performance issues.[8]

Two types of parallel algorithms derive from this approach:

a. An algorithm that creates and destroys tasks dynamically and uses load balancing when mapping the tasks to the processors, or

b. A SIMD program with one task assigned to each processor. In this case, the stages of agglomeration and mapping can be unified.

2.5.1 Partition

The partition phase’s main goal is to identify the points where parallelisation opportunities occur, in other words, find the tasks that can be executed in parallel. In this phase, we try to identify as many tasks that can be parallelised, but this decision can be reconsidered in a later phase.

Two techniques can facilitate the partitioning phase:

a. Domain decomposition: Aiming to divide the data in relatively equal chunks and then partition the computation, associating it to the data it operates on. The main focus here, is on large data structures, or frequently accessed data, or

b. Functional decomposition: Dividing the computation first and proceeding to the data requirements. If data requirements from different parts of the computation are overlapping, we need to establish communication to avoid replicating data.

2.5.2 Communication

In the communication phase, we try to identify the channels required to establish communication between tasks. Communication can be described based on three features.

Local: Communication only with the immediate neighbours (producer-consumer relation), or
Global: Communication with every other task in the grip, in which case producer-consumer pairs are not suitable.

Structured: Communicative tasks form regular patterns, such as trees or grids, or
Unstructured: Communication forms an irregular and unstable grid.

Synchronous: Both communicating partners are aware of the time that communication is required, or
Asynchronous: Consumer has to explicitly request data.

2.5.3 Agglomeration
The Agglomeration phase, handles the combination of the refined tasks, specified in the Partition phase, in a way to achieve efficiency. The tasks are grouped in less and larger sized sets. The primary goal is to increase granularity and consequently reduce costs of communication and task creation. Having a small number of tasks also reduces the volume of overall communication, as it gives the possibility of replication of data instead. This consolidation of tasks needs to be irrespective of specific architecture to maintain flexibility. Finally, development costs must be taken into consideration, in the case that the design of the algorithm imposes changes to existing cooperating systems. If the resulting number of tasks, from this phase, is equal to the number of processors, the mapping phase can be omitted.

2.5.4 Mapping
The mapping of tasks to processors does not occur on uni-core machines or shared memory architectures, as they provide automatic task scheduling. When it comes to distributed architectures, mapping is considered a trade-off between executing tasks on different processors to address concurrency and executing on the same processor to increase locality.

Load balancing techniques are a decisive aspect of mapping, generally based on domain decomposition, they aim to assign a coarse-grained task to each available processor. On the other hand, task scheduling algorithms, based on functional decomposition of the partition stage, represent a task pool, distributed or centralised, from which tasks are chosen for allocation to processors.

2.6 Problem Application
In order to compare the parallel performance of the three languages, we have searched for an algorithm that combines a set of specific features. Firstly, it needs to be simple, easy to implement in three different languages, considering the relatively short timeframe of the project. Secondly, it needs to be data intensive, to justify the need for parallelisation, and also well-tuned, to
facilitate the adjustment to the requirements of the project. Finally, we aim to a widely used algorithm, as this would indicate a significant amount of research previously done on it.

2.6.1 Field of interest
The N-body family of algorithms is used in a range of scientific applications, such as simulation of astrophysical stellar systems [36] and molecular dynamics [37]. In astrophysics, the N-body algorithm is used to measure the gravitational interaction among stars, which governs the evolution of black holes and the creation of planetary systems [36]. Figure 2.6 A two dimensional simulation of two colliding galaxies [36] shows consecutive screenshots of the simulation of a two dimensional galaxy collision [37] [38].

![Figure 2.6 A two dimensional simulation of two colliding galaxies [36]](image)

Due to the macroscopic nature of gravity, in order to calculate the force received at each mass, all other masses in the system must be taken into consideration. The calculation of all the masses is called cross product. Eventually, as the number of bodies, N, increases, the amount of calculations of the cross product scales to $N^2$. Due to the large complexity of the problem, it is very difficult to achieve high accuracy on a single workstation [36].
From the point of parallel computing, N-body problems are an important category of parallel algorithms, calculating interactions among large sets of data. The computationally intense nature of these problems is compatible to the goals of parallel programming. The effort to achieve optimal solutions to these problems is justified by their wide use on both macroscopic (galaxies) and microscopic (molecules) scale.

2.6.2 Specification

2.6.2.1 The Problem

The problem concerns the calculation of the orbit of the masses. Given a set of N points (either 2D or 3D), their masses and their velocity-vectors, compute the gravitational interactions and adjust the velocity vectors. The problem is solved iteratively (for T iterations) in small time steps by calculating the force of gravity of each mass and then calculating the motion of each mass during this short time [39].

Every mass in the system receives gravitational traction from any other mass, in proportion to the inverse square of the distance between the two objects. From Newton’s law of gravity we know that the force between two objects is given by the following equation:

$$F = G \frac{m_1 \cdot m_2}{d^2}$$

where

$G$ is the gravitational constant

$m1$ and $m2$ are the masses of the objects

and $d$ is the distance between them.

Figures 2.G and 2.H below, show the forces acting on an object $m1$ and the calculation of the force by another object $m2$, in a two-dimensional space [39].
2.6.2.2 The all-pairs algorithm

The high complexity and irregularity of the N-body problem, shows the prohibitive cost of direct summation for large N’s [37], when performed sequentially. This direct summation technique is referred to as the all-pairs algorithm. This algorithm is considered a benchmark application to indicate the performance achieved by different programming languages, both new and old. We have chosen to implement this algorithm in the three novel languages, as an adequate measure of the performance they claim to achieve.

2.6.2.3 The Barnes-Hut algorithm

Other hierarchical algorithms have been proposed for faster summation of all the forces, which maintain, under some restrictions, the requested high accuracy. A widely used technique is the Barnes-Hut Algorithm, originally published in 1986 by Josh Barnes and Piet Hut [38]. The algorithm uses grouping of objects to reduce the number of calculations and “far away” groups of bodies are reduced to one interaction with their center of mass. In the Barnes-Hut algorithm the space is recursively subdivided, to the point where each subdomain contains only one object. The groups’ center of masses, are represented by quad-trees for two-dimensional spaces or by oc-trees for three-dimensional spaces.
3 Small Scale Language Assessment

3.1 Tools & Compilers

Chapel

Chapel is supported on Cray systems but it also portable to most unix-based parallel platforms, like SGI Altix, IBM Power Series and commodity clusters as the one used for this project (see 4.1 Experiment setup details). Also Chapel is portable to desktop machines running UNIX, MacOS X and Windows with Cygwin. Such a machine (laptop PC) was used for first implementations of the code and debugging.

Chapel’s target code is C and C++. The prerequisites include an environment that supports linux commands as cd, mkdir, rm etc, a Bourne-shell and C-shell and env that can locate Perl and Python on the system. Finally, access to gmake or other GNU -compatible make and awk and standard C and C++ compilers are required. One single Chapel installation can support multiple compilers and platforms, since all binary files are stored in CHPL environment variables.

Chapel programs use a launcher which wraps job startup. It performs command line parsing prior to starting the parallel execution, to avoid errors or typos. Also, it provides the convenience of launching a program using one single library, without having to specify number of nodes, number of cores per processor etc and coordinates I/O. The basic supported launcher is GASNet’s amudprun for execution over UDP, but there are also other available implementations. Also, the latest release supports five implementations of tasks, among them fifo, which is the commonest, and qthreads.

The installation of Chapel was straightforward. A tarball is available in the download link of the official webpage[40]. The tarball contains the latest release of the compiler (version 1.7.0) with the standard supporting libraries, a short instruction manual for quick installation, example programs, the language specification (version 0.93) and some further documentation concerning the latest changes, a review of the status of the language and the user agreement. Furthermore, there exist some specific README files for building the compiler on other platforms, multi-locale execution instructions and details on how Chapel maps tasks to threads.

There is also a newer version of the compiler, available on the svn repository, but since this is a work in progress, we did not choose to develop our programs based on this unstable version. According to the webpage, the next version of Chapel (1.8.0) is expected to be released on October 2014.
X10

X10 uses source-to-source compilation to another language, which is executing using its platform specific tools. Version 2.3.1 is compiled to C++ or Java. In the first case, the resulting program is compiled by the C++ compiler resulting to Native X10. In the second case, the program is compiled to class files and executed on JVMs (Managed X10). The platforms available are: AIX/Power, BlueGene/P, Linux/Power, Cygwin/x86, Linux/x86 and x86_64, and Mac OS x86 and x86_64.

The X10 compiler (version 2.3.1) is available for downloading in the webpage[41], controlled by the Eclipse Public Licence v1.0 [42]. They also provide pre-built binaries specific to each supported platform. The pre-built binaries contain the compiler with the standard libraries, sample programs and installation instructions. This distribution contains the X10RT libraries, which handle inter-process communication using either sockets (default), SMP via shared memory or MPI and other implementations for transport protocols, like PAMI and DCMF.

There is also the X10DT, licenced by the Eclipse Plugin Licence 1.0, which is a plug-in for the Eclipse suite and contains Eclipse, the compiler and the X10 runtime. The problem with this tool is that it is only supported for a fairly old version of Eclipse( Eclipse 3.7.1 Indigo) and the plug-in crashes very often. Although, this was considered a useful development tool, in the beginning of the project, it turned out to be inconvenient. Apart from the compiler, the developers’ team provides separate downloads for their test suite and the benchmarks they are using.

The installation process for X10, lasted longer than expected, as we first chose to install the X10DT, in which we developed the preliminary code examples. The pre-built version, on the other hand, was easy to install, except maybe from the amount of dependencies. For this purpose, we consulted the detailed setup instructions file, also pointed from the main page of the language. In general, it took about one week to complete this procedure.

UPC

The UPC Berkeley compiler (version 2.16.2) is available in the language’s download page [43]. The system’s requirements for the installation are minimal. A POSIX-like environment, a Unix version or Windows with Cygwin, and a new version of a C compiler. Optionally: an MPI implementation, to run UPC over MPI and a C++ compiler to enable execution over UDP.

UPC is translated to C (front-end compiler) and compiled as a C program (back-end compiler). We have chosen to point to the default UPC-to-C translator which is accessed over the Internet, for the front-end. This solution is convenient and also supports a range of platforms on which the UPC compiler does not build. As it is highlighted in installation instructions, the back-end compiler needs to be stable and to continue working properly even after the installation of UPC, since the compilation in C has an impact on UPC program’s performance. UPC’s executables are compiled with respect to the particular network API. Available network API’s are the following: UDP, MPI, DCMF and SMP (see [43] for a detailed list of supported API’s).
For the development of UPC programs, the debugging and the testing we have used the already installed compiler in the school’s lab. For assessment purposes only, we installed UPC on a laptop running Linux (Ubuntu version 12.04). In the initial configuration, we only had to specify where the C and C++ compiler and the path to the MPI implementation. Afterwards, we have tried to add profiling capabilities. The Global-Address-Space Profiling (GASP) [44] interface is supported in UPC, which enables to link UPC to third party profiling tools. Although we have managed to enable GASP, linking to the proposed tool (Parallel Performance Wizard [45][46]) has not been successful.

3.2 Current State

Chapel

Chapel’s web page points to a number of referenced papers and publications. The papers are listed both as overviews and presentations of Chapel’s features [47][48]. Also, there exist some research papers evaluating the language and exploring performance issues [49]. All these papers are recent (dating back to 2009). Chapel is also referenced in a big amount of older research papers concerning mostly parallel performance coding (like MPI and OpenMP implementations). These papers usually refer to Chapel in their Future Work section, or more generally to the potential performance gain arising from the PGAS programming model and languages.

As far as the community is concerned, as it is expected for a research language, the mailing lists are busy. The “users’” mailing list, has provided some valuable information for the completion of this task, both reading other users’ enquiries and also posts of the author. Also, the “bug reporting” and “developers’” lists are checked on daily basis and lots of fixes that require better understanding of the compiler are promptly handled.
Other than that, it seems that Chapel has gained the interest of many developers of parallel computing (the large number of enquiries and clarifications indicates this fact, see Figure 3.A). The users are often prompted to write their own functions, on unresolved issues and contribute to the development process.

The STATUS file provided in the installation zip of Chapel refers to a number of known issues and bugs related to the implementation. Some of the main fallbacks that have had an impact on the project’s implementation are referred bellow:

**General issues**

- User-defined constructors, as the Body constructor, are not robust. There have been some problems with referencing and dereferencing the values.
- Using a comparison operator with a record and ‘nil’ results in an internal compiler error, unless the un-initialised records are set to “nil” manually.

**Multi–locale execution**

- Continue statements in forall and coforall statements not implemented. If a continue statement was available, then the whole program could be written using one coforall loop and a continue statement would be used to distinguish between single- and multi- locale execution
- Parameter ranges are not supported, that is the reason why the myInds workaround is used in the initialisation of the arrays on each locale.

**Domains and Arrays**

- Input of whole arrays is not implemented.
- Arrays of arrays where the inner arrays vary in size are not supported. Due to this, each locale is assigned a fixed size chunk of data (Body objects), with some of the positions set to nil.
- Sparse domain/array slicing is not supported. This has mainly affected initial implementations of the Laplace function in which he have tried to use a “chessboard” iteration of the distributed domain.

**Standard Distributions and Layouts**

- Blocked distributed arrays with bounding boxes that do not overlap with the bounds of the domain cannot be assigned.
- Load balancing in distributions is not implemented properly.

**Task Parallelism and Synchronization**

- Atomic statements are not yet implemented.
- Deadlock may occur due to insufficient number of threads.
• Sync and single type qualifiers are not disallowed on types that are not supported. The sync variables used in the program were meant to be declared of type Body at first, and the compiler did not complaint about it.

**Data Parallelism**

• Data parallel statements are serialized with a warning message "x has been serialized".
• Assignments from ranges to multidimensional arrays are always serialized.
• Array promotion/forall/for/scan expressions always evaluate to 1D arrays.
• Parallel (forall) domain iteration ignores alignment of the ranges used to declare the domain. Before the declaration of distributed array as array of arrays, many exceptions occurred when trying to access objects which were allocated on other locales.

**X10**

The number of publications concerning X10 is large and constantly updated. Only this year there have been listed eleven research papers (pointed from X10’s web page [41]). These research papers are based on the growing interest in the language and concern various domains and experimentation, like applications, performance and concurrency research, tool development and libraries & frameworks.

Applications concern the use of X10 and the exploitation of its characteristics (eg. type safety) in fields as cloud computing, visualisation infrastructures, graph analysis, parallel Dynamic Programming (DP) and financial computing. Current tool development efforts focus on testing tools, optimisation and verification techniques, debuggers and performance monitoring. Library development refers to data clustering algorithms and their enhancements and also to the adaptation of already existing frameworks to X10. Finally, studies concerning performance and scalability highlight the importance of race-free programs, which are deterministic and preserve atomicity of operations. For an overview of these projects, see “X10 in use” section in [41].

The growing interest of the scientific community for X10 is also indicated by the increasing number of high-profile universities, among them UCLA and Columbia University, that choose to integrate X10 to their program of studies [41] and the yearly X10 workshops, intemented as forums for programmers and developers. Finally, the dedicated mailing lists, answer users’ equiries promptly (see Figure 3.B).
The X10 project developers use the JIRA platform [50] for bug reporting and issue tracking. There are a number of features described in the specification that do not currently work and are intended to be fixed in subsequent releases. One such bug is the X10 runtime’s use of “busy wait” loops in the worker threads that execute asyncs. This means that in a place that has no asyncs to execute, one worker thread will still consume CPU cycles, waiting to receive network messages. A summary of known issues is presented below:

- Currently there is no way to configure machines with multiple IP addresses
- All X10 classes are required to preload by Managed X10
- There is no dynamic constraint check for accesses to global reference
- Exceptions fail if the methods of a class are in the wrong order.

**UPC**

UPC is the most popular, in terms of community interest, of the three languages assessed in this project. As a result, the amount of publications concerning the language is slowly decreasing while the language is reaching the stage of adoption by many programmers. The existing publications analyze communication and runtime issues, discuss compiler optimisations and application development, while research concerning performance analysis or design issues is covered in older publications. Most recent publications refer to communication tuning for bottleneck avoidance [51], data race detection [52] and applications in the field of linear algebra [53].
As far as the community’s activity, mailing lists dedicated to UPC are different to the other languages’, discussed previously. Here the mailing list is mostly used for announcements (conferences, calls for papers etc) rather than enquiries, clarifications and problems in development. This can be attributed to the fact that the language is adopted by a wider audience and thus a variety of documentation sources and tutorials are available to the users.

A status review of UPC is provided in the main webpage, pointing to known bugs and limitations. Most of them are due to the translation to C and incompatibilities of C compiler. For example, it is stated that the optimisation flags of gcc4 are not allowed with shared local access, and shared memory optimisations only work with Pthreads. Others, are related to the development of UPC programs. For example, assembly code cannot be inlined in UPC code although the programmers can include it in C header files or the fact that stdin, stdout and stderr are unsafe to use, as reading and writing from different threads can be interleaved.

3.3 Ease of learning – Qualitative Assessment

Chapel

The learning phase of this project started with Chapel. Its object oriented features, are very similar to Java and C++, a fact that reduced the learning difficulty. On the other hand, task parallel features (like the begin statements) have been used in the past, for example in OpenMP code. The general structure of programs and the main special features are explained in detail in numerous tutorials pointed to from Chapel’s web page. This initial approach lasted only five days.

As far as the specific characteristics are concerned, like distributions and domain slicing, we have spotted some inconsistencies between the specification and the available tutorials. These refer mainly to the syntax of declarations of distributions. Also, the specification lacks detail on the definition of distributions and layouts. Due to the nature of the problem application, distributed arrays were essential to the implementation. As a result, the lack of detailed documentation caused delays during the first three weeks of development.

X10

X10’s sequential core is very intuitive for a programmer who has worked with high-level object oriented languages. Since, X10 programs are primarily translated to Java (Managed X10), which is also one of the languages X10 is based on, the only problems relate to syntactical differences. Especially, the classes that did not require parallel constructs (Body and myRandom) were written in very short time and were not changed till the final version.
Concerning the parallel constructs of the language and the rough start of Chapel, the learning procedure here was faster. The fact that X10 parallelises all loops by default, also gave some immediate speedups for our preliminary implementations. In the part of the distributions, the declaration and the fact that all distributed arrays require initialisation stalled the development procedure.

Also, although X10 was the first fully developed version of the problem application, we did not get any speedups, mainly due to the big amount of communication and we had to restructure the code to achieve a better trade-off between communication and computation phases. These changes were proposed by the developers, after an enquiry about lack of speedup, in the mailing list.

**UPC**

UPC was the last language we started but also the fastest to complete. Due to the better understanding of the algorithm from the previous two languages, debugging has been made much easier. On the other hand, UPC’s execution model, though very different to Chapel and X10, shows similarities to previously used parallel technologies like OpenMP.

Shared arrays and layouts are simple and straightforward to implement. The parallel constructs, like upc_forall, were already familiar from Chapel. The difficulty here was to understand how to structure the communication and the coordination between threads, in order to simulate a “ring” topology of the running threads. Finally, UPC gave speedups without any major changes in the code.
4 Large Scale Language Assessment

4.1 Experiment Setup Details

The Beowulf cluster, which was used for performance testing, is composed of 32 nodes, each with 8 cores. The nodes are of type NUMA with 4096kB L3 cache and 256kB L2 cache memory. Each core’s local IDE disk space is 5.7 GB. The workstations run Linux CentOS Release 6.4 with 3.7 GiB of RAM. Each of the processors is of type Intel (R) Core ™ i7.

4.2 Discussion on the all-pairs N-body code

4.2.1 General structure

We have tried to use object oriented features wherever possible in the implementations, to separate the core functionality from the auxiliary code.

For the development of the N-body application in Chapel, we have used three separate files:

- Body.chpl, which implements the Body class, the constructor and the helper functions
- myRandom.chpl, which implements the random generator and
- uniComm.chpl, where the main computation is performed

Chapel supports object oriented programming, but also since it is meant to be an HPC language, it is highly based on functions. This, along with the applications simplicity, has given the advantage of implementing the main computation and the calling function in the same file, avoiding multiple function calls and object instantiation. For the full source listings see Source Listings section 1, in Appendix.

In X10, there are four files in total:

- Body.x10, implementing the Body class,
- myRandom.x10, implementing the random generator
- UniverseC.x10, where the main calculation takes place and
- GenBlock.x10, which is a caller function containing main()

X10 is highly object oriented and encourages the use of clean interfaces and the separation of tasks. The caller function (GenBlock.x10), as used here, separates the calculation from the helper functions and reflects the program’s order of execution. The source files for the X10 implementation are provided in Source Listings, section 2 of the Appendix.

Finally, in UPC, we only use one file:
first.c for the calculation

As UPC is a variation of C, there are no classes or objects. The Body class of the other two languages is implemented as a C struct here. The program’s different tasks are separated in functions which are called in the according order by the main function, also contained in first.c. UPC’s source code is available in Source Listings, section 3 of the Appendix.

4.2.2 Distribution & domain maps

High performance applications require means to distribute geometric data. Distributions are such high level concepts, which are provided in all three languages.

In Chapel we have used the Block distribution, which is defined in the BlockDist module. The Block distribution partitions the indices of a domain in blocks using a bounding box argument and maps them on the available locales. A two dimensional domain (8x8) will be mapped on 6 locales as below:

```
0 0 0 0 1 1 1 1
0 0 0 0 1 1 1 1
0 0 0 0 1 1 1 1
2 2 2 2 3 3 3 3
2 2 2 2 3 3 3 3
2 2 2 2 3 3 3 3
4 4 4 4 5 5 5 5
4 4 4 4 5 5 5 5
```

Figure 4.1 A Block-distributed domain (8x8) on 6 locales

In the code, we used the following distributed domain:

```c
const D = {0..(numBodies-1)}; //domain of total bodies
const ddom = D dmapped Block(boundingBox=D); //distribution of domain D
```

In X10, we used the Unique distribution which according to the language’s API, creates a distribution that maps every point of the region in a different place and maps at least one point to every place. The Block distribution offered in X10’s default distributions, was harder to implement and we got some scrambled results when assigning the bodies array. This distribution performs an implicit mapping of virtual shared data to the underlying distributed memory. It is tunnable to the degree of specifying the number of locales and it employs data balancing.

Finally, in UPC we used the blocked layout, which is one of the defaults. This layout is represented by “[*]” in the declaration of the array. Due to the fact that, in UPC, shared array sizes need to be multiples of the total number of threads (THREADS) defined in execution time with “-n
#, we do not need to perform manual chunking of the arrays. In essence the chunk, equals the integer multiplied with THREADS (see below). An example of block distributed arrays is shown below:

```chapel
#define N 100*THREADS
shared
t int [**] v1[N], v2[N], sum[N];

void main() {
 int i;
 upc forall(i=0; i<N; i++; &sum[i])
 sum[i]=v1[i]+v2[i];
}
```

Figure 4.8 Example of vector addition using blocked layout[34]

4.2.3 Arrays

The arrays’ declarations in Chapel is as follows:

```chapel
var bodies:[ddom][ch] Body; //the distributed array of bodies
var otherBodies:[ddom][D][ch]Body; //The distributed array which stores the bodies of remote locales
var syncs:[ddom] sync int; //distributed array of sync variables
```

where `ch` is a non-distributed domain declared as

```chapel
var ch = [1..chunk];
```

Chunk is calculated as:

```chapel
var chunk = numBodies / numLocales; //chunk size assigned to each locale

if (mod(numBodies, numLocales)>0){
  chunk = chunk+mod(numBodies, numLocales);
}
```

The Block distribution employs data balancing on the distributed domain, but this is a static assignment of size chunk, which maps more bodies to each locale, thus affecting the random initialisation of the distributed array.

The array could not be initialised and passed by a caller function without iteration over locales, because only the values of the first locale were passed. Considering that, we needed initialisation to be deterministic for all executions and produce the same input values we had to find some workaround.

As a first solution, we considered reading the input values from files but we did not manage to accomplish that task, mainly due to the lack of example code. Chapel uses channels, similarly to Java, to open a file, but the only material we had in our disposal was the specification of the language and there was no clarification on how to combine the existing functions. Other than that, reading from a file on a distributed environment needs to take care of interleaving reads from different threads on different locales, probably delaying the initialisation.
The workaround we finally chose, was to initialise a non-distributed array of the required dimension (numBodies) and then identify the indices each locale owned. The following code snippet uses a coforall loop to iterate in parallel over locales. Each locale identifies the indices it owns, using myInds, which was provided in a post in the mailing lists. Then, we iterate over myInds and copy the contents of the initial array (init) to a non-distributed array bodiesHere. bodiesHere is of size chunk which means that it may contain null values. Finally this array is copied back to the distributed array in the position here.id, which indicates the id of the executing locale.

```java
coforall loc in Locales do
  on loc {
    const myInds = ddom._value.locDoms[here.id].myBlock;
    var bodiesHere:[ch] Body;
    var i:int = 1;
    for gridPoint in myInds{
      bodiesHere(i) = init(gridPoint);
      i+=1;
    }
    bodies(here.id) = bodiesHere;
  }
```

In the above part of the code, we iterate over myInds, meaning the indices each locale owns, copying the Body objects from the init array to the local array bodiesHere. When bodiesHere is initialised, we copy it back to the bodies distributed array in position here.id which is the index of the current locale.

In X10, the distributed arrays’ declaration is the following:

```java
private val bodies : DistArray[Rail[Body]](1);
private val otherBodies : DistArray[Rail[Rail[Body]]](1);
```

and inside the Universe class constructor we have:

```java
this.bodies = DistArray.make[Rail[Body]](Dist.makeUnique());
this.otherBodies = DistArray.make[Rail[Rail[Body]]](Dist.makeUnique(), (p : Point) => new Rail[Rail[Body]](Place.MAX_PLACES));
```

For performance reasons we have used Rails instead of Arrays. Rail is a special case of an array, which represents a generic, zero based, rectangular, one dimensional bound checked and continuous memory array structure with indices of type long (public type Rail[T] = Array[T] {self.rank==1,self.zeroBased,self.rect,self.rail};).

As it is often pointed by the developers in the mailing list, bound checking for arrays is expensive and the Rail type instead of one-dimensional arrays helps the X10 optimiser gather information on how to improve performance. In the next release of the language, Rails will become a distinct class, separate from arrays.
We have used the same workaround as above to initialise the array of bodies, only here, we iterate over the distribution of places, using a finish block, and explicitly calculate the number of bodies that we needed to map to each place. The code is the following:

```java
finish ateach(place in Dist.makeUnique()) {
    val startHere = here.id * chunkSize;
    val endHere = Math.min(numBodies, (here.id+1) * chunkSize);
    val numBodiesHere = Math.max(0, endHere-startHere-1);
    val bodiesHere = new Rail[Body](numBodiesHere+1);
    var i:Int = 0;
    for(gridPoint in startHere..(endHere-1)) {
        bodiesHere(i) = init(gridPoint);
        i++;
    }
    bodies(here.id) = bodiesHere;
}
```

In UPC the array declaration is the following:

```upc
shared [*] struct Body bodies[numBodies];
shared [*] struct Body otherBodies[THREADS][chunk];
```

where:

- `THREADS` is the total number of threads executing and it is specified during execution with --n
- `numBodies` is declared as a simple multiple of `THREADS` and
- `chunk` is this multiple (an integer number).

The initialisation of the array is straightforward, compared to the previous two languages. Here, we iterate over the distributed array, produce the random values and instantiate the bodies. The reference to the array position (&(bodies[i])), allows only to the owner thread to operate on each object. The use of pointers, indicates the low level coordination that UPC employs.

```upc
void init(){
    int z = 2001;
    for (int i=0; i<numBodies; i++){
        double r = myRandom(z);
        shared struct Body * b = (shared struct Body *)&(bodies[i]);
        b->mass = next(r);
        b->posx = next(b->mass);
        b->posy = next(b->posx);
        b->posz = next(b->posy);
        b->velx = next(b->posz);
        b->vely = next(b->velx);
    }
}
```
In Chapel, the declaration of distributed arrays is identical to the declaration of non-distributed arrays. In that sense Chapel hides the aspect of distribution from the array data type. The distribution affects the domain of the array. On the other hand, in X10 and UPC, distributed arrays differ syntactically to non-distributed.

Another difference is UPC’s limitation of lengths. The distributed arrays’ sizes need to be multiples of the total number of threads of the program. Although, THREADS are an execution parameter, this requirement is inconvenient for testing.

### 4.2.4 Locales’ / places’ / threads’ communication

The algorithm we used performs two phase communication and calculation. In the first communication phase, every locale sends its part to the next locale. Then they all calculate the interactions among their bodies. When the calculation is finished they receive a part from the previous locale, send their part to the next one and compute the interactions between the two parts. This second communication phase occurs inside a while loop which is accessed until all locales have send their parts to all the other locales (until the next locale to receive from, is the current locale) [54]. The figure below, gives an overview of the algorithm.
This algorithm doubles the distance for exchanging data in each communication phase. Since, each node sends and receives in each phase (apart from the first, that everyone calculates their own interactions) starting from the current node id, and adding or subtracting one, it is guaranteed that all nodes will parse all other nodes and exchange data with everyone.

In Chapel, communication between locales is performed by indexing the global array `Locales[]` and simulating a ring topology (last sends to first etc). In the first communication phase, we use integer arithmetic to find the next locale’s id and then we use the “on” clause to switch to that locale and copy the array toSent.

**Communication phase 1**

```chapel
// before starting computation, send my bodies (toSent) to the next place
var toSent :[ch] Body = myBodies;
var me = here.id;
var nextPlace = here.id+1;
if(nextPlace == numLocales) { nextPlace=0;}
if (nextPlace!= me ){
    on Locales[nextPlace]{
        otherBodies(nextPlace)(me) = toSent;
        syncs(nextPlace) = 1;
    }
```
In the second communication phase, each locale waits to receive the chunk of bodies from the previous phase by indexing the source. The source’s id will indicate the position from which the locale will read, in the otherBodies distributed array. The communication only occurs when the total number of locales is bigger than one (first conditional) and only after all local computation is finished. Afterwards, the global exchange of data starts and the sync variable in the position here of the syncs array is checked. The value of the variable is less important here, meaning that the variable will be read after it is written (by the source). This conditional operates as a barrier, to avoid reading null values. After the receipt, each locale proceeds and sends the toSent array to the next target. This occurs inside a while loop, which will exit when all locales have sent their chunk to all other locales and the next target becomes the current locale (target == here.id).

Communication phase 2

```plaintext
//receive
var target = nextPlace + 1;
if(target == numLocales) {target = 0;}
var source = me - 1;
if(source < 0) {source = numLocales - 1;}

if (numLocales != 1) {
    if (syncs(me) == 1) {
        if (debug) {writeln("PROCEEDing.. I am ", here.id);}
    }
}
while (source != me) {

    if (target != me) {
        // send myBodies (toSent) to the next target place
        on Locales[target] {
            otherBodies(target)(me) = toSent;
        }
    }
    //calculation phase 2 ...

    target = target + 1;
    if (target == numLocales) {target = 0;}
    source = source - 1;
    if (source < 0) {source = numLocales - 1;}
```

X10’s implementation is very similar to Chapel’s. The first difference we can comment, is that copying from array to array of the same size is not straightforward as in Chapel. In essence here, we declare the toSent array and we need to initialise it. Therefore, we instantiate new objects for each array position and initialise them with the values of the according field in the source array.

Code Part 4.I Chapel - Communication Phase 1

Code Part 4.II Chapel - Communication Phase 2
Another, difference is that places, in contrast to locales, are implemented in a ring topology. This means that no bound checks are needed, instead we use the functions next() and previous() of the class Place, included in the standard library of the language. In order to write to a remote position, we use the “at” construct and we enclose copying in an atomic block to ensure that the access is not interrupted.

Communication phase 1

```java
val toSent = new Rail[Body](myBodies.size as Int, (i:Int)="new
Body(myBodies(i).mass,myBodies(i).posx,myBodies(i).posy, myBodies(i).posz,
myBodies(i).velx, myBodies(i).vely, myBodies(i).velz));

// before starting computation, send my bodies (toSent) to the next place
val nextPlace = here.next();
if (nextPlace != here) {
   @Uncounted at(nextPlace) async {
      atomic {
         otherBodies(nextPlace.id)(pl) = toSent;
      }
   }
}
```

Code Part 4.III X10 - Communication Phase 2

The most important difference, occurring in the second communication phase is the use of X10’s “when” construct. “When” is checking the condition in the parenthesis until it is fulfilled. Afterwards, the place proceeds to the execution of the next statement. This construct is valuable in this distributed environment, because it works as a barrier for each place/thread, but in a more convenient way, without stalling the execution on other places.

Communication phase 2

```java
var target : Place = nextPlace.next();
var source : Place = here.prev();
while (source != here) {
   if (target != here) {
      // send myBodies (toSent) to the next target place
      val targetPlace = target;
      @Uncounted at(targetPlace) async {
         atomic {
            otherBodies(targetPlace.id)(pl) = toSent;
         }
      }
      // wait on receipt of a set of bodies from other place
      when(otherBodies(here.id)(source.id) != null);
   }
   //calculation phase 2..
```
target = target.next();
source = source.prev();

Code Part 4.IV X10 - Communication Phase 2

UPC’s implementation is very similar to Chapel’s. Again, here we need to perform bound checks for the source and the target locale and we use integers for indexing. The THREADS constant is also an integer.

One of UPC’s differences is the need to iterate over the target and source array element by element when copying, whereas Chapel and X10 provide bulk operations on arrays (and sub-arrays). Also, we use the while loop with a reference to the otherBodies to check when it is ready to read. As the NULL value applies to objects, and not arrays of objects, we choose to check the last position in the array (the last Body). Testing has shown that writing is sequential, thus when the last becomes non empty, all other positions are already written.

Communication phase 1

//copy myBodies to toSent
for (int j=0; j<chunk; j++){
    toSent[j] = myBodies[j];
}

// before starting computation, send my bodies (toSent) to the next place
int me = MYTHREAD;
int next = me+1;
if(next==THREADS){next=0;}

if (next!= me ){
    for (int j=0; j<chunk; j++)
    {
        otherBodies[next][j] = toSent[j];
    }
}

Code Part 4.V UPC- Communication Phase 1

Communication phase 2

int target = next + 1;
int source = me -1;

if(target==THREADS){target=0;}
if(source<0){source=THREADS-1;}

while (source != me) {
    if (target != me) {
        // send myBodies (toSent) to the next target place
        for (int j=0; j<chunk; j++)
        {
            otherBodies[target][j] = toSent[j];
        }
    }
}
// wait on receipt of a set of bodies from other place
shared struct Body *ot = (shared struct Body *)&(otherBodies[me][chunk-1]);

if (ot == NULL){

    while (&(otherBodies[me][chunk-1]) == NULL){}

}

//computation phase 2
target = target+1;
if(target ==THREADS){ target=0;}
source = source -1;
if(source<0){source = THREADS-1;}

Code Part 4.VI UPC - Communication Phase 2

4.2.5 Computation

The computation is performed with iteration over the locales/places/threads in all three languages. More specifically, in the first computation phase, where each body computes their own chunk, we iterate over the lower triangular part of the myBodies array. Since each interaction involves a pair of bodies, we can then update the velocity of both bodies involved in each iteration.

In the second computation phase, where we calculate interactions between the myBodies and otherBodies arrays, we need to iterate over all pairs, calculating only the new velocity of the owned bodies.

The core computation is identical in all three languages. We calculate the distance of the two bodies for the three dimensions and if this distance is not zero then we update the velocity either for both or one of the bodies, depending on the computation phase (one or two). Afterwards we call the energy function and, we calculate the energy of the interaction and add it to the total energy produced in the current place. In the second computation phase we add the energy of the interaction divided by two, since we only update our own body.

```c
var dx: real = bodyI.posx - bodyJ.posx;
var dy: real = bodyI.posy - bodyJ.posy;
var dz: real = bodyI.posz - bodyJ.posz;

var d2: real = dx*dx + dy*dy + dz*dz;
//update my bodies' velocity
if (d2 != 0.0 ) {

    var mag: real = dt/ (d2*sqrt(d2));
    //only updates the positions of myBodies
    bodyI.velx -= dx* bodyJ.mass * mag;
    bodyI.vely -= dy* bodyJ.mass * mag;
    bodyI.velz -= dz* bodyJ.mass * mag;
```
var e3 = energy2(bodyI, bodyJ, d2);

energyThisPlace += e3/2;
}

The energy calculation uses two distinct functions. The first one, named energy1 is used in the first computation phase and it takes one body as argument. It calculates the initial energy of each body, obtained by its initial velocity. The second function, named energy2, calculates the energy of the interaction between two bodies. Its arguments are the two bodies and the distance between them. This second function, is used in both computation phases. The implementation of the energy function is simple and thus very similar in the three programs. Below, we provide the two implementations in Chapel.

```chapel
//calculates the energy between two bodies residing on the same locale
proc energy1(b:Body):real {
    return 0.5 * b.mass * (b.velx*b.velx + b.vely*b.vely + b.velz*b.velz);
}

//calculates the energy between two bodies residing on different locales
proc energy2(b1:Body, b2:Body, d:real):real {
    return (b1.mass*b2.mass)/sqrt(d);
}
```

### 4.2.6 Language specific features

In order to iterate over the locales in Chapel we use the `coforall loop` construct over the `Locales` array. A new thread is created in each iteration (in essence, one thread per locale), which handles the communication and the computation. An example of a coforall loop, employing task parallelism, is shown below.

```chapel
config const numTasks = here.numCores;
coforall tid in 1..numTasks {
    writeln("Hello, world", " from task ", tid, " of ", numTasks, "!");
}
```

**Code Part 4.7Coforall loop example[55]**

In the N-body code we have used

```chapel
coforall loc in Locales do
    on loc {..}
```

In X10 we used the `ateach` construct to iterate over the distributed array of bodies. Similarly to Chapel’s coforall loop, all iterations execute in parallel, in the specified by the distribution place. The ateach statement is equivalent to the following:

```chapel
for point p : D.region async (D[p]) S
```
Ateach is designed to facilitate parallel matrix code independently of the declared distribution [56]. An example of its use is shown below:

```java
public class TutAteach1 {
    public static void main(String args[]) {
        finish ateach(point[i] : dist.factory.unique() ) {
            System.out.println("Hello from " + i);
        }
    } // main()
} // TutAteach1
```

In our code, we use: `ateach(pl in bodies)` to iterate over the distributed array.

In UPC, we iterate over the total number of threads (THREADS) using the `upc_forall` construct. `upc_forall` spawns an independent thread for each iteration and helps exploit data locality [57]. The syntax of the `upc_forall` loop is `upc_forall(init; test; loop; affinity)` where affinity is either an integer or a reference to a shared object.

In our code, we use the following loop `upc_forall(int i=0; i<THREADS; i++; i)`, to iterate over all threads, with affinity to the `i`th integer, corresponding to the `i`th thread.

One of the features specific to Chapel is the iteration on objects. Apart from the regular iteration over indices ranges, and in Chapel’s case over domains, it is possible to iterate over the values/objects of a non-distributed array with a for loop. The three cases are shown below.

```java
// iteration over a range
for i in 1..myInds.size {
    var bodyI = myBodies(i);
    //..
}

// iteration over domain
for j in ch2 {
    var bodyJ = myBodies(j);
    //..
}

// iteration over value/object
for mb in myBodies {
    var body = mb;
    //..
}
```

Also we need to mention that Chapel uses `nil` instead of null to denote empty objects. Lastly, the `myInds` construct (see 4.2.3 Arrays), has proven useful for the purposes of the development and we hope to see it formally introduced in the following releases.
A specific feature of X10, is the distinction between \emph{vals}, which behave as constants inside the block they reside and \emph{vars}, which behave as regular variables.

Also, in the matter of the energy calculation, X10 does not allow different places to access the same variable. To solve this problem, we have used a helper struct named SumReducer, which was provided by the developers. We have combined ateach with the \emph{finish} construct, which waits on the termination of all the threads created by ateach. At the end of the ateach loop each place contributes the energy calculated, to the SumReducer, using the \emph{offer} construct. The SumReducer struct (see Code Part 4.X) performs a simple evaluation of the sum of all the calculated energy values over all the places in the distribution.

```scala
val directEnergy = finish(mySumReducer()) {
  ateach(pl in bodies) {
    var energyThisPlace: Double = 0.0;
    //calculation
    offer energyThisPlace;
  } //ateach
}
//energy
return directEnergy;
```

\textbf{Code Part 4.X X10 - Energy calculation}

Finally the \texttt{@Uncounted} annotation specifies that the enclosing finish block of async (in this case, ateach) does not need to wait for completion at the synchronisation point [58].
Table 4.A shows the declaration of constructs, used in the implementation, that have similar functionality in the three languages.

<table>
<thead>
<tr>
<th>Chapel</th>
<th>X10</th>
<th>UPC</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loops</td>
<td>Forall, coforall</td>
<td>upc_forall</td>
<td>Spawn one thread for each iteration</td>
</tr>
<tr>
<td>Locale/place/thread shift</td>
<td>on</td>
<td>at</td>
<td>shared int *p; (pointer to shared) Shift to remote place in the distribution</td>
</tr>
<tr>
<td>Distributions</td>
<td>domain</td>
<td>makeBlock(Region r)</td>
<td>Blocked distribution</td>
</tr>
<tr>
<td>Domain</td>
<td>dmapped Block</td>
<td>makeCyclic(Region r, Int axis)</td>
<td>Cyclic distribution</td>
</tr>
<tr>
<td>iteration over distributions</td>
<td>coforall (over Locales)</td>
<td>ateach</td>
<td>upc_forall (over THREADS) Iteration over the distribution (locales/places/threads)</td>
</tr>
<tr>
<td>Locale/place/thread id</td>
<td>here.id</td>
<td>here</td>
<td>MYTHREAD Id of the current locale/place/thread</td>
</tr>
</tbody>
</table>

Table 4.A Language constructs with similar functionality in Chapel, X10 and UPC

4.2.7 Development issues

One of the main issues we encountered during development, was the lack of speedup in Chapel and X10. Our initial version of the code, was using the provided parallel constructs of the two languages, without explicit communication between locales/places. More specifically the code in Chapel used two nested forall loops iterating over the distributed array of bodies. X10’s implementation used two nested finish(ateach..) loops. This setup did not produce any speedups.

After some research, and discussion with the developers and other users via the mailing lists, we realised that the communication between locales/places was stalling the program. More specifically we were producing a big number of messages containing two little information. For example, in X10, the ateach loop switched to the target place to get one specific body and returned to the calculation. This gives complexity of numBodies*numBodies for the communication, which is far more expensive than the actual computation.

As a result, we were pointed by X10’s developers to employ lower level coordination. We added the second array (otherBodies) to avoid interference between places/locales. This resulted in the final version of the algorithm in X10. In order to get safer conclusions on performance, we decided to use the same algorithm in all three languages. The need to change the code (only X10 and Chapel version were complete at that point), stalled the development procedure for almost three weeks.
Furthermore, after restructuring the algorithm, we kept getting poor performance from Chapel’s testing, as shown in section 4.3, and also we have not been able to run the program with the input sizes we used in the other two languages. The programs crashes after the input of 1200 bodies. After some discussion with the developers, we discovered that the otherBodies array is not assigned, due to its large memory requirements (see 4.3.2 - Chapel).

Furthermore, we were aiming to perform testing on both distributed and shared memory configurations. A special feature in Chapel’s implementation is that it distinguishes shared from distributed memory and uses different constructs to address it. For distributed memory, Chapel uses distributions while for shared memory on one node, it uses layouts. Furthermore, the developers have focused their main efforts on distributions, stalling the development of layouts. This concludes to three facts. There is no documentation or examples about the default layouts. Secondly, the layouts are not fully implemented for all available types of arrays. Lastly, there is no association between the predefined layouts and distributions (block, cyclic etc). Chapel’s lack of support for layouts is analysed in more detail in section 4.3.2.

Another issue with X10, that we have not yet resolved, is the long time we needed to take the measurements when using large inputs. For example, for an input of 100.000 bodies executing on one place, we had to wait for almost 14 hours. On the other hand, the execution time was minor (milliseconds) compared to the user time. We have not yet resolved, why this is happening but monitoring of the cluster has shown that the X10 program, requires a lot of time to spawn slave processes and remains in this state for long. When all slave processes are spawned, the calculation is very fast.

For UPC, we faced problems during the testing phase. The testing was performed manually (not by launching a long job from a script file) due to the static threads compilation mode. THREADS (the total number of running threads) need to be specified at compile time, as program constant, because it defines the length of the shared array. In order to produce comparable inputs to the other languages testing, we had to change the values (i.e. perform the chunking manually) and recompile every time.

4.2.8 Code quality

In terms of code quality, we could commend on the use of classes/structs and functions. We have used them wherever possible in our implementations. The X10 version of the program is the most generic as expected, due to the object oriented basis of the language. The classes Body, Generator and myRandom and also the energy calculation function are independent modules. As for the advance function, it is mainly oriented to solve the specific problem, although the interchanging communication and computation phases are a common pattern in parallel development. Exception handlers, is an important feature we are aiming to add to the X10 version, to enhance the code and make it more general.
Chapel’s version is more straightforward, there is no caller function and the main function resides in the computation file. Classes (like the Body class) were used wherever possible, but since Chapel is a language built from first principles its programs do not follow strict object oriented style. Also, the workarounds we used (like myInds) cannot be considered general, since they are not formally included in the latest release.

Finally, UPC, taking under consideration that it is a variation of C, does not make use of many generic features. The functions we used to structure the program are in a way independent but not general enough to address other applications. A probable fix would be to restructure the printing of arrays in a function.
4.3 Performance Comparison

4.3.1 Speedup graphs

4.3.1.1 Distributed memory

4.3.1.1.1 Chapel

For the experiments in Chapel we have set each node on the cluster to represent one locale, thus each node uses one thread.

![Chapel runtime graph](image)

Chapel’s runtimes are irregular and the program, generally, does not scale. For the smaller inputs, we notice a minor decrease in the runtime, when running from two to four locales. Even in this case, it is much slower than the execution on one locale. Other than that, the program is slowing down, when adding more locales. The runtimes of execution on one locale (sequential performance) are smaller than any other parallel setup. This poor performance could be attributed to synchronisation and communication costs.
Speedup reaches a peak value of 1.2 to 1.6 when executing on four locales. Also, there is a minor increase when changing from 10 to 12 locales, for all input sizes.

4.3.1.2 X10

In the following experiments, one place is represented by one node on the system. The hostfile, we are passing as argument, contains twelve separate nodes and for every execution we set X10_NPLACES= #, while X10_NTHREADS remains 1.
In the graph, we notice an exponential increase in runtimes when doubling the input from 8000 to 16000 bodies, for execution on one (sequential) and two places. The same occurs when increasing the input from 16000 to 24000 bodies.

From the runtime table we can see that small inputs (1000 and 4000 bodies), give minor speedups. On the contrary, bigger inputs give a speedup factor close to five between execution on one and twelve places (see Figure 4.G X10 speedup graph). Especially, for the input of 24000 bodies, the speedup factor for execution on twelve places reaches 5.41, which gives an efficiency rate of 0.45.
4.3.1.3 UPC

For UPC, we used the smallest possible number of 8-core nodes. Here, one thread corresponds to one core. For example, for 10 executing threads we use two nodes, for 50 threads we use seven nodes etc. We have compiled the code using UDP as the underlying network as it is the recommended setting for performance studies.
The runtime graph in UPC reveals irregularities. Execution with small inputs, 1000 to 4000 bodies, with the use of two threads is slightly slower than execution with one thread. Especially for input of 8000 bodies, the runtime almost doubles.

For larger inputs, 16000 and 24000 bodies, we achieve a minor speedup when executing with two threads. Also, runtime is decreasing for execution with 50 threads.

On the other hand, when we use 100 worker threads, runtime increases independently of the input size. For 16000 bodies, for example, execution with one hundred threads gives similar runtime to the execution with ten threads.
The speedup graph of UPC is also irregular. Speedup is increasing for four threads with a speedup factor of 1. For small inputs, we see a stable or decreasing speedup when adding more threads. For the input size of 16000 bodies, we achieve speedup of 1 when the number of executing threads changes from 10 to 50. Finally, for 24000 bodies, we get a speedup of approximately 1.5, when running with 20 and 50 threads.
4.3.1.2 Shared memory

4.3.1.2.1 X10

In our experiments using shared memory, we only pass as argument the hostfile with one entry. We increase number of places (X10_NPLACES), from one to eight, for our 8-core machine. Here places represent cores.

The runtime graph shows the program’s consistent behaviour for all input sizes. Execution on two places is slower than on one place. After increasing the places to four and eight we get speedup, which is shown below.
X10’s speedup graph is decreasing for the input size of 1000, while the speedup factor for input of 24000 bodies on 8 places reaches 5.87. All the intermediate input sizes give a small speedup.

4.3.1.2.2 UPC

In the following experiments we pass the number of nodes N equal to one, and we increase the number of processes (threads) up to 8. We have compiled the UPC program with pthreads to achieve better performance for SMP.
The runtime graph in UPC indicates a slowdown when running with two threads independently of the input size. For four and eight executing threads, the runtime decreases up to 50%.

UPC’s speedup graph is identical for all input sizes. This seems legitimate, as in UPC each thread executes an instance of the program in SPMD mode.
4.3.2 Discussion on performance

Chapel

In Chapel, for the distributed memory configuration, we performed measurements of smaller inputs than in the other two languages. The reason is that, bigger inputs did not produce any results. When we tried inputs of 1200 bodies or more, the program crashed.

On the other hand, these bigger inputs have given results when run on a larger number of locales. More specifically, each locale, is able to handle up to 1000 bodies and after this limit it fails to execute. For example, an input of 4000 bodies, will work when run on 4 locales or more.

This problem is due to the large data structures we assign in the program’s beginning. Especially the otherBodies array will have size of \( \frac{num\text{Bodies}^3}{num\text{Locales}} \) bodies \( (\text{numBodies} \times \text{numBodies} \times \frac{\text{numBodies}}{\text{numLocales}}) \). For the algorithm we use, the otherBodies array is required to have such a length, since it is used to store all the other locale’s bodies on each locale. For execution on one locale and input of 1200, the array will contain 1,728,000,000 bodies, so the locale runs out of memory and the execution stops.

Chapel’s coforall loop creates a task for each iteration, where a task is a logical parallel computation for execution via a thread. This means that, for a “coforall i in 1..1000” loop, we create 1000 tasks. These tasks run on the current locale, unless they are migrated to another locale using an “on” clause. In our case, we have used a coforall loop over the number of available locales (up to twelve in our experiment). Thus, we can assume that the problem is not caused by the consumption of the available threads of the cores.

Although, the algorithm we used is designed to hide communication latency, by interchanging communication with computation, it seems that Chapel does not benefit from this setup. The required set up to achieve this explicit communication has a negative impact on execution times.

In addition, we do not present any measurements for shared memory configuration. Chapel does not address shared and distributed memory setups with the same language constructs. More specifically, it uses distributions for distributed memory and layouts for shared memory. These two constructs, although they have the same functional purpose; to handle the mapping of processes to the underlying hardware, are implemented differently. The current support for layouts in Chapel is limited and as stated in the mailing lists, the focus of developers is on distributions. This fact is also obvious by the lack of documentation, from examples in tutorials to the actual section of the specification which is empty.

X10

In X10, although the input sizes are relatively small, we managed to get good performance results. For the execution on distributed memory, we got a speedup factor close to 5 between executions on one and on twelve places, with peak value of 5.41 for the larger input. The speedup is
increasing more rapidly, when the input is larger. Some additional measurements for larger inputs can be found in the Appendix.

In the shared memory execution, the best speedup was calculated to 5.87. The graph below compares the speedup of the executions on both memory models using up to 8 cores, for the input size of 24000 bodies.

![Graph showing speedup comparison for 24000 bodies input]  
**Figure 4.J Speedup comparison for 24000 bodies input –X10**

The previous figure corroborates that communication over distributed memory is more expensive than on shared memory. Especially, for execution on eight places the speedup on shared memory exceeds the corresponding value on distributed memory by a factor of 1.57.

**UPC**

The runtime and speedup graphs in UPC for the execution on distributed memory show irregularities. The peak performance is achieved for the larger inputs (16000 and 24000 bodies) when executing with 50 threads. One possibility for the downturn of speedup on 100 threads, is the granularity of tasks. Another possible cause for these could be the high workload on the cluster during testing.

On shared memory set up, we managed to achieve good speedup for four and eight executing threads although there is a fallback for two threaded execution. The speedup graph, on the other hand, indicates the SPMD mode of UPC. The next figure depicts the speedup achieved for 24000 bodies input, when executing with 2, 4 and 8 threads on one node (shared memory) and two nodes (distributed memory).
The above figure shows the results of two and four threads executed on one node for distributed and shared memory. As mentioned earlier, in the distributed memory testing we have tried to use the least possible nodes. In this sense, the irregularity is due to the different underlying network. For the shared memory execution we used pthreads, while for the distributed memory we have used UDP for the underlying network.

In the last case, of 8(shared) and 10 threads (distributed) we see that shared memory, even with less executing threads, performs better than 10 executing threads on two nodes. There are two possible reasons for this behaviour. The first and most obvious reason is the communication cost of distributed memory. The second reason could be task granularity which decreases when distributing the same input on 8 and 10 threads.
The following two graphs show the speedup achieved for input of 24000 bodies in X10 and UPC. This input has given best speedup for both implementations. We cannot provide any comparative results for Chapel, since the larger input we were able to execute was 1600 bodies.

**Figure 4.1 Comparative Speedup for 24000 bodies - Distributed Memory**

Figure 4.1 Comparative Speedup for 24000 bodies - Distributed Memory shows the speedup on distributed memory configuration. The speedup for UPC is an approximation to the number of used nodes, as we specified the number of threads for the tests. The table shows the actual number of nodes used in each case. From this graph, X10 appears to scale better and achieves better speedups.

<table>
<thead>
<tr>
<th>Threads</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
</tr>
<tr>
<td>50</td>
<td>7</td>
</tr>
<tr>
<td>100</td>
<td>13</td>
</tr>
</tbody>
</table>
Figure 4. M Comparative Speedup for 24000 bodies - Shared Memory

Figure 4. M Comparative Speedup for 24000 bodies - Shared Memory shows the speedup of the two implementations for X10 and UPC, on shared memory configuration on one node, where X10, gives a better speedup factor than UPC.
5 Conclusion

5.1 Summary

The project has attempted a detailed assessment of Chapel, X10 and UPC, including:

Chapter 2 gives a comparative survey of the main language features and constructs of PGAS languages emphasizing on the parallel novel constructs they implement.

Chapter 3 covers the small scale assessment, looking at programmability and pragmatics. In particular, it looks at availability and ease of use of software, development tools and compilers. Also, we look at the community’s activity, measuring mailing lists’ posts, publications and other evidence indicating that there is an active interest on each language. Finally, we attempt a qualitative assessment on the ease of learning, which depends on the author’s programming background.

Chapter 4 covers the large scale assessment, the development of the all-pairs N-body code in the three languages. We discuss code differences of the same algorithm on the three languages. Also, we provide speedup graphs and we comment on the parallel performance of the final programs both on distributed and shared memory configurations.

5.2 Evaluation

5.2.1 Language evaluation

The project has provided a basis for comparison on the different aspects of the three languages. Looking back to the Small Scale assessment (Chapter 3) we have provided details on compilers and tools, current state and ease of learning. In terms of supporting tools, all three languages give the opportunity for quick installation without major problems. UPC probably has the most demanding installation, since it requires the specification of paths to the back-end compilers and decision on the underlying communication network right from the beginning, while there are many prerequisites in order to make full use of the languages capabilities (eg.in order to use UDP as the underlying network a C++ compiler is required).

In terms of current state, we could argue that most unresolved issues occur in Chapel, but this could be attributed to the fact that Chapel is designed from first principles, thus many of the core features are implemented without some other language’s sound basis. On the other hand, UPC is passing to the stage of adoption, meaning that more people get to use it and subsequently bugs and limitations are easier to locate and fix.

About the ease of learning, according to the author’s view, programming background and experience from this project, we could say that X10 is probably the quickest to pick up. It is highly based on object oriented features and facilitates code writing. Also, it provides a rich API library which helps locating dependencies between library classes and functions. Chapel’s sequential core
was also easy to learn but the parallel constructs like the distributions required more effort. UPC required a totally different approach, but the similarities to OpenMP and some previous experience with explicit thread handling helped reducing the learning curve.

Concerning parallel performance, it is obvious from Section 4.3 that X10 achieved the best results both in shared and distributed memory configurations. In this part, we should acknowledge the fact that the final algorithm we used is based on an X10 implementation of the N-body problem[54]. The same algorithm was then implemented in the other two languages. As a result, we had expected that the X10 version would already be optimised at some level. Considering that the algorithm employs low level coordination and UPC uses thread-level coordination, it was expected that results in UPC would be satisfactory too. On the other hand, Chapel’s performance is far from satisfactory. This has been a surprise to us, during the testing phase, since Chapel is developed to address performance and productivity. As pointed in the future work section, we are aiming to develop a program more adapted to Chapel’s parallel constructs and philosophy and compare the performance results of the two implementations.

As a bottomline, our suggestion to a new parallel programmer, aiming to familiarise to the PGAS languages, would be to start with X10. Provided that there exists a sound object oriented background, X10 gives the opportunity for experimentation on parallel programming at high level. Although, keeping in mind that very often, a programmer will resort to low level solutions in order to achieve performance.

5.2.2 Project Evaluation

The project involved some challenges from the very beginning, since the languages we chose to assess are in research stage and not yet standardised. We identified some risks during our initial research and, in many cases, we had to employ mitigation strategies. The first and almost immediate risk we faced, concerned the set of the assessed languages. Our initial aim, was to assess Chapel, X10 and Fortress by Sun Microsystems. Fortress, was one of D.A.R.P.A’s HPCS program final products, but its active development stopped due to complications with the language’s type system on existing virtual machines, as stated in the forum of Oracle labs. We then switched to UPC, and we had to perform the initial research of features and tools for the new language. Also, we reordered the learning and development phases in the three languages, starting with Chapel.

The second risk, identified as “Tutorial and data availability”, also had to be mitigated with task rescheduling. Especially, during the first weeks of developing in Chapel, we tried to understand how some of the basic new constructs were implemented internally (like distributions and layouts). The documentation lacked detail, so we had to address the mailing lists. While waiting for answers, we switched to X10’s development and vice versa. So the initial planning of completing the code in one language and starting the next one, was neither feasible nor convenient, and we had to adapt the time plan.

Stability of software was another issue that delayed the development slightly. In X10, we have been looking at sample code, adapted to the upcoming release of the compiler. This code did
not comply, in terms of syntax and constructs, with the current release. At that point, we chose to keep developing with the stable release and restructure the code instead of experimenting with an unstable compiler, in order to avoid further risks in the project.

Finally, resource availability, has been a major problem during the last weeks before the submission deadline. The load on the Beowulf cluster, has been such, that our results were irregular and unstable. As a last minute solution, we tried to rerun some of our tests, but with reduced number of executions.

We have to admit that we have not been able to follow the initial task plan strictly and that after the learning phase we kept switching between tasks. Also, our estimate of one month for the implementations, proved insufficient. During the development phase, we chose to proceed with the all pairs algorithm, although our intention was to use the Barnes-Hut algorithm. The development phase, postponed all following phases, and especially the write up, which started two weeks before the deadline.

In general, looking at the requirements analysis, we have fulfilled most of the requirements. The evaluation of different algorithms for the N-body problem (see 1.3.2 I) is one requirement we did not address due to time limitation, but considering that this is a assessment of languages, we needed the same algorithm in all three languages and not necessarily the most tuned and advanced algorithm for the problem application. Performance tuning, as part of requirement II, was another task we did not perform, although the restructuring of the algorithm and other minor optimisations, which can be considered as tuning, were employed, wherever possible during the development phase.

5.3 Contribution

The contribution of this work can be summarized in the following:

- We provide a comparative review of features and language constructs, emphasizing on the parallel constructs, the abstraction mechanisms and the communication and placement control employed by each of the languages.
- We review the status of the languages, based on the amount of publications and community support. Also, we discuss the availability of learning materials and the ease of installation.
- We provide implementations of the all-pairs algorithm for the N-body problem which is an intensive benchmark application, in the three under assessment languages.
- We assess important pragmatic issues such as the differences in the produced code, comparing the used constructs in 3 state-of-the-art PGAS languages.
- We provide a set of measurements and we elaborate on performance issues, both on shared memory and distributed memory configurations.
5.4 Future Work

We aim to extend this work in the future and use it as a starting point for PhD level. Some of the main points for future work are summarized below. Some of them are immediate and concern bug fixes or additional testing, while others set the aims for further research on PGAS languages and parallel programming, in general. We aim to:

- perform additional testing and tuning of the produced programs. This was not performed earlier due to time limitations and the cluster’s high workload. We aim to use as many cores as possible to extend our comparison.
- implement a new version of the N-body code in Chapel, using an algorithm without explicit communication
- investigate the problems of long user times in X10 and Chapel’s behaviour of killing processes.
- add exception handling wherever possible
- perform additional testing of the UPC version, with other available compilation options (pthreads and MPI)
- restructure the code in all three languages to use other available distributions or layouts and comment on performance and differences
- develop different problem applications (like Laplace function) oriented to task parallelism and measure performance
- develop the same algorithm in Titanium and Co-Array Fortran, and extend our comparison to include all PGAS languages
Bibliography


[34] K. Yelick, “Partitioned Global Address Space Programming with Unified Parallel C (UPC).” UC Berkeley.


Appendix
Source Listings

1. Chapel

Body.chpl

/*
 * Implements the Body class and contains the constructor and some helper functions
 *
 * Author: Konstantina Panagiotpoulou
 */

use Random;

class Body {

    var mass: real; // body's mass
    var posx, posy, posz : real; // body's position on x, y, z axis
    var velx, vely, velz : real; // body's velocity on x, y, z axis

    // default constructor
    proc Body(){
        mass = 1.1;
        posx = 1.1;
        posy = 1.1;
        posz = 1.1;
        velx = 1.1;
        vely = 1.1;
        velz = 1.1;
    }

    // constructor
        this.mass = mass;
        this.posx = posx;
        this.posy = posy;
        this.posz = posz;
        this.velx = velx;
        this.vely = vely;
        this.velz = velz;
    }

    // helper functions
    proc getMass(): real{ return mass; }
    proc setMass(mass: real){this.mass=mass;}

    proc getPosX(): real{ return posx; }
    proc setPosX(posx: real){this.posx=posx;}
}
proc getPosY(): real {return posy;}
proc setPosY(posy: real){this.posy=posy;}

proc getPosZ(): real {return posz;}
proc setPosZ(posz: real){this.posz=posz;}

proc getVelX(): real {return veltx;}
proc setVelX(veltx: real){this.veltx=veltx;}

proc getVelY(): real {return velty;}
proc setVelY(velty: real){this.veltvy=velty;}

proc getVelZ(): real {return veltz;}
proc setVelZ(veltz: real){this.veltz=veltz;}

//printing function
proc printBody()
{
     writeln("mass: ", this.getMass(), "|| X: ", this.getPosX(), "|| Y: ", this.getPosY(), "|| Z: ", this.getPosZ(), 
     "\n velx: ", this.getVelX(), "|| vely: ", this.getVelY(), "|| velz: ", this.getVelZ() );
}

//checks if two bodies are equal
proc equals(b: Body): bool
{
     if (this.mass == b.getMass() && this.posx==b.getPosX() &&
         this.posy==b.getPosY() && this.posz==b.getPosZ()) then
         {return true;}
     else {return false;}
}
}
uniComm.chpl

/*
 * Implements the main class which performs the interaction and energy calculations
 * Author: Konstantina Panagiotpoulou
 */

use Time;
use Body;
use Random;
use Math;
use BlockDist;
use myRandom;

config var numBodies: int = 10; //total number of bodies in the universe
config var numIterations: int = 120; // number of iterations
config var debug, debug2, verbose : bool = false; //debuging and printing

controls
var dimensions: (real, real); //dimensions of the universe
var dt: real = 0.1; //interval of iterations

const D = {0..(numBodies-1)}; //domain of total number of bodies
const ddom = D dmapped Block(boundingBox=D); //Block distribution of domain D

var chunk = numBodies /numLocales; //calculates the chunk assigned to each locale

if (mod(numBodies, numLocales)>0){ chunk = chunk+mod(numBodies, numLocales);}

var ch = {1..chunk};

var bodies: [ddom][ch] Body; //the distributed array of bodies

var otherBodies: [ddom][D][ch]Body; //The distributed array which stores the bodies of remote locales

var syncs: [ddom] sync int; //distributed array of sync variables

var tinit: Timer;
tinit.start();
    //initial non-distributed array initialised with random values
var init: [D] Body;

d var z:int = 2001;
for i in D {
    var r = new myRandom(z);
    var ms = r.next();
    var px = r.next();
    var py = r.next();
    var pz = r.next();
    var vx = r.next();
    var vy = r.next();
    var vz = r.next();

    init(i) = new Body(ms, px, py, pz, vx, vy, vz);
    z+=2;
}

//executes on all locales in parallel
coforall loc in Locales do
    on loc {
        //returns the indices owned on the current locale
        const myInds =
            ddom._value.locDoms[here.id].myBlock;

        var bodiesHere:[ch] Body;

        var i:int = 1;
        //iterates over myInds and copies the values from
        the initial array
        for gridPoint in myInds{
            bodiesHere(i) = init(gridPoint);
            i+=1;
        }

        bodies(here.id) = bodiesHere;
    }
tinit.stop();

//prints the initial positions of the bodies in the universe
if (verbose){
    writeln("initial positions");
    forall loc in Locales do
        on loc {
            var bo = bodies(here.id);
            for b in bo{
                if (b!=nil){
                    b.printBody();
                    writeln("---IN PLACE: ",
                        here.id, " with name : ", here.name);
                }
            }
        }
}

//starts the timer
var t: Timer;
var energy = 0.0;
t.start();
/executes on all locales in parallel
coforall loc in Locales do

var energyThisPlace= 0.0;
const myInds = ddom._value.locDoms[here.id].myBlock;

var mydom = (0..(myInds.size -1));
//copies the bodies of the current locale to a local array
var myBodies = bodies(here.id);
//prepares a local array to sent to the other locales
var toSent :[ch] Body = myBodies;

if (debug2){
    writeln("MYBODIES from : ", here.id);
    for mb in myBodies{
        if(mb!=nil){
            writeln("I am: ",here.id);
            mb.printBody();
        }
    }
    writeln("MYTOSENT from : ", here.id);
    for mb in toSent{
        if(mb!=nil){
            writeln("I am: ",here.id);
            mb.printBody();
        }
    }
}

// before starting computation, send my bodies (toSent) to
// the next place

var me = here.id; //identifies the ciurrent locale
var nextPlace = here.id+1; //calculates the next one
if(nextPlace ==numLocales) { nextPlace=0; } //bound check

    var mydd = mydom;
    if (nextPlace!= me ){
        //shifts to the
        nextPlace and writes the toSent array
        on Locales[nextPlace]{
            otherBodies(nextPlace)(me) = toSent;
            //assigns a value to the sync
            syncs(nextPlace) = 1;
        }
    }
// calculates the energy for all interactions within this place

for i in 1..myInds.size {
    var bodyI = myBodies(i);
    var en1 = energy1(bodyI);

    energyThisPlace+=en1;
    var ch2={1..(i-1)};
    for j in ch2 {
        var bodyJ = myBodies(j);

        var dx: real  = bodyI.posx - bodyJ.posx;
        var dy: real  = bodyI.posy - bodyJ.posy;
        var dz: real  = bodyI.posz - bodyJ.posz;

        var d2: real  = dx*dx + dy*dy + dz*dz;
        // updates my bodies' velocity
        if (d2 != 0.0)
        {
            var mag: real  = dt/(d2*sqrt(d2));
            bodyI.velx -= dx*bodyJ.mass * mag;
            bodyJ.velx += dx*bodyI.mass * mag;
            bodyI.vely -= dy*bodyJ.mass * mag;
            bodyJ.vely += dy*bodyI.mass * mag;
            bodyI.velz -= dz*bodyJ.mass * mag;
            bodyJ.velz += dz*bodyI.mass * mag;

            var e2 = energy2(bodyI, bodyJ, d2);
            energyThisPlace+= e2; // + ???
        }
    }
}

// if there is only one locale, update mybodies' position
if (numLocales ==1){
    for i in 1..myInds.size {
        var bodyI = myBodies(i);
        bodyI.posx += dt* bodyI.velx;
        bodyI.posy += dt* bodyI.vely;
        bodyI.posz += dt* bodyI.velz;
    }
}

// calculates the new target and source locale and performs bound checks
var target = nextPlace + 1;
if(target ==numLocales){target=0;}
var source = me - 1;
if(source<0){source = numLocales-1;}
// waits to receive from source locale
if (numLocales==1){
    // checks sync variable
    if (syncs(me) == 1){
if(debug){writeln("PROCEEDing.. I am ",
here.id);};

/*he while loop terminates when the current locale has
* locales in the universe and the next source locale is
* itself
*/

while (source != me) {

    if (target != me) {
        // send myBodies (toSent) to the
        next target place
        on Locales[target]
        otherBodies(target)(me)
        = toSent;
    }

    // calculates all interactions with otherBodies at
    other place
    for j in ch{
        var bodyJ= otherBodies(me)(source)(j);

        if (bodyJ !=nil){
            // checks if the position is empty
            for i in 1..myInds.size{
                var bodyI = myBodies(i);

                var dx: real = bodyI.posx -
                bodyJ.posx;

                var dy: real = bodyI.posy -
                bodyJ.posy;

                var dz: real = bodyI.posz -
                bodyJ.posz;

                var d2: real  = dx*dx + dy*dy +
                dz*dz;

                // update my bodies' velocity
                if (d2 != 0.0 ) {
                    var mag: real  = dt/
                    (d2*sqrt(d2));

                    // only updates the
                    positions of myBodies
                    bodyJ.mass * mag;
                    bodyJ.mass * mag;
                    bodyJ.mass * mag;

                    bodyJ, d2);
                    +=e3/2;

                    energyThisPlace
                    +=e3/2;
                }
            }
        }
    }
}
// if (bodyJ != nil)
// for j in ch{

// update my bodies' position

for i in 1..myInds.size {
    var bodyI = myBodies(i);
    bodyI.posx += dt * bodyI.velx;
    bodyI.posy += dt * bodyI.vely;
    bodyI.posz += dt * bodyI.velz;
}

// calculates the new target and source locale

if (target == numLocales) {target = 0;}
source = source - 1;
if (source < 0) {source = numLocales - 1;}

energy += energyThisPlace; // contributes the energy of the place to the total energy
}

// prints the final positions of the bodies in the universe
if (verbose) {
    writeln("final positions");
    forall loc in Locales do
        on loc {
            var bo = bodies(here.id);
            for b in bo{
                if (b != nil) {
                    b.printBody();
                    writeln("---IN PLACE: ", here.id, ", with name : ", here.name);
                }
            }
        }
    }

writeln("Total energy: ", energy);
writeln("Time elapsed: ", t.elapsed());
writeln("Time initialisation: ", tinit.elapsed());

// calculates the energy between two bodies residing on the same locale
proc energy1(b: Body): real {
    return 0.5 * b.mass * (b.velx * b.velx + b.vely * b.vely + b.velz * b.velz);
}

// calculates the energy between two bodies residing on different locales
proc energy2(b1: Body, b2: Body, d: real): real {

```chpl
return (bl.mass*b2.mass)/sqrt(d);
}

myRandom.chpl

/*
 * Implements a random generator for double numbers
 *    * Author: Konstantina Panagiotpoulou
 */

use Math;

class myRandom {
    var seed: int;
    var x: real;

    proc myRandom(seed:int){
        this.x = seed;
    }

    proc next(): real{
        x = mod((0.456*x + 0.34), 65386.0);

        return x;
    }
}
```
2. X10

GenBlock.x10

/*
 * Implements the caller class which generates the universe
 * and calls main() function
 * Author: Konstantina Panagiotopoulou 2013
 */

import UniverseC;
import Body;
import myRandom;

import x10.util.Timer;
import x10.io.Printer;
import x10.io.Console;
import x10.util.*;

class GenBlock{

    static val bodies: Int = 4; //total number of bodies in the universe
    static val x: double = 3.5; //dimension x of the universe
    static val y: double = 4.2; //dimension y of the universe
    static val dt:double = 0.1; //interval of iterations
    static val numIterations: Int = 120; //total number of iterations
    static val verbose :boolean = true; //printing control

    static val B:Region(1) = 0..bodies;

    public def run(bodies: Int, verbose:boolean){

        //calculates the chunk size for each place
        val chunk: Int = (bodies / Place.MAX_PLACES) + ((bodies %
Place.MAX_PLACES > 0) ? 1 : 0);

        //instantiates a UniverseC object
        val uni = new UniverseC(x, y, bodies, chunk);

        //starts the timer
        val fill = new Timer();
        val startfill: long = fill.milliTime();

        //calls fillUniverse function
        uni.fillUniverse(chunk, bodies);

        //stops the timer
        val stopfill: long = fill.milliTime();

        //prints the initial positions of the bodies in teh universe
        if (verbose){

            //print the initial positions

        }else{

            //print the final positions

        }

    }

}

94 | P a g e
```java
Console.OUT.println("==== Initial positions ====");
uni.printBodies(); // bo

var e: double = 0.0;
var d : Double = 0.0;
// starts a new timer
val t = new Timer();

val start: long = t.milliTime();
// calls the advance function numIterations times and sums up its result in d (total energy)
for (var i: Int = 1; i <= numIterations; i++) {
    d += uni.Advance(dt);
}
// stops timer
val stop: long = t.milliTime();
// prints final positions of the bodies in the universe
if (verbose){
    Console.OUT.println("==== Final positions ====");
    uni.printBodies();
}
// calculates elapsed time
val total:long = stop-start;
val filltotal:long = stopfill - startfill;

// prints elapsed time and total energy
Console.OUT.println("Total energy : "+d);
Console.OUT.println("\n");
Console.OUT.println("Time elapsed: "+((total as double)/1e9));
Console.OUT.println("Time for initialisation: "+((filltotal as double)/1e9));

// the main() function of the program
public static def main(args: Array[String](1))
{
    val nb = Int.parse(args(0));
    val ver = Int.parse(args(1));
    var v: boolean;
    if (ver==0){ v = false;}
    else{v = true;}

    // instantiates a new GenBlock object and calls run function
    new GenBlock().run(nb, v);
}
```

UniverseC.x10
import x10.util.Random;
import Body;
import x10.io.Printer;
import x10.io.Console;
import x10.lang.Math;
import x10.array.DistArray;
import x10.util.*;
import x10.compiler.Uncounted;
import x10.util.Team;
import x10.lang.Reducible.*;

public class UniverseC {
    public var dimx: double;  //dimension x of the universe
    public var dimy: double;  //dimension y of the universe
    public var numBodies: Int;  //total number of bodies in the universe
    public var e: double;  //energy
    public val dt: double = 3;  //interval of iterations
    public var chunk: Int;  //chunk size

    //distributed array of bodies
    private val bodies: DistArray[Rail[Body]][1];
    //distributed array of arrays of bodies
    private val otherBodies: DistArray[Rail[Rail[Body]]][1];

    //constructor
    public def this(dimx: double, dimy: double, numBodies: Int, chunk: Int){
        this.dimx = dimx;
        this.dimy = dimy;
        this.chunk = chunk;
        /* The bodies array is distributed over the available places
         * using the Unique distribution, which assigns each element to a
         * different place and
         * every elements to one place
         */
        this.bodies = DistArray.make[Rail[Body]](Dist.makeUnique());
        /* The otherBodies array is distributed over teh available places,
        * using the Unique distribution and contains non distributed
        * arrays
        * of size Place.MAX_PLACES
        */
this.otherBodies = DistArray.make[Rail[Rail[Body]]]
(Dist.makeUnique(), (p : Point) => new Rail[Rail[Body]]
(Place.MAX_PLACES));
}

/*Initialises the distributed bodies array
 * using the non-distributed array init
 */
public def fillUniverse(chunkSize: Int, numBodies: Int){

val init = new Rail[Body](numBodies);
var z:Int = 2001;
//generates random values for the elements (bodies) of the array
init
for (i in 0..(numBodies-1)){
    val r = new myRandom(z);
    val ms = r.next();
    val px = r.next();
    val py = r.next();
    val pz = r.next();
    val vx = r.next();
    val vy = r.next();
    val vz = r.next();

    init(i) = new Body(ms, px, py, pz, vx, vy, vz);
z+=2;
}
/*maps the elements of the distributed array to the
 * elements of init and copies their values,
 * iterating over the distribution
 */

//executes in parallel at each place in the distribution
finish atEach(place in Dist.makeUnique()) {

    //calculates where to start reading in the initial array
    val startHere = here.id * chunkSize;

    //calculates where to stop reading the initial array
    val endHere = Math.min(numBodies, (here.id+1) * chunkSize);
    //calculates the total number of bodies this locale owns
    val numBodiesHere = Math.max(0, endHere-startHere-1);
    //declares a local array
    val bodiesHere = new Rail[Body](numBodiesHere+1);
    var i:Int = 0;

    //iterates over the initial array, inside the calculated range and
    copies to the local array
    for(gridPoint in startHere..(endHere-1)) {

        bodiesHere(i) = init(gridPoint);
        i++;
    }
    //assigns the contents of the local array to the distributed array

bodies(here.id) = bodiesHere;
}
}

/* The advance function performs the communication and
the calculation of the new velocities and positions of the bodies in the
universe
and returns the produced energy from the interactions */
public def Advance(dt:double):double{
/* The SymReducer module sums up all
local energies calculated at each place */
val directEnergy = finish(mySumReducer())

//executes in parallel for each place
ateach(pl in bodies) {
  //myBodies: local copy of the distributed array with the current place's elements
  val myBodies : Rail[Body] = bodies(pl);

  //toSent: copy of myBodies for exchange with the other places
  val toSent = new Rail[Body](myBodies.size as Int, (i:Int)=>new
  Body(myBodies(i).mass,myBodies(i).posx, myBodies(i).posy, myBodies(i).posz, myBodies(i).velx, myBodies(i).vely, myBodies(i).velz));

  var energyThisPlace: Double = 0.0;
  //calculates the next place
  val nextPlace = here.next();

  /* Communication between places is performed by writing in the
  otherBodies distributed array
  in the position of the target place */

  if (nextPlace != here) {
    @Uncounted at(nextPlace) async {
      atomic {
        otherBodies(nextPlace.id)(pl) = toSent;
      }
    }
  }

  //calculates the interactions within this place
  for (i in 0..(myBodies.size-1)) {
    val bodyI = myBodies(i);
    val enl = energyI(bodyI);
    energyThisPlace+=enl; //\ldots
    for (j in 0..(i-1)) {
      val bodyJ = myBodies(j);
      val dx: double = bodyI.posx - bodyJ.posx;
      val dy: double = bodyI.posy - bodyJ.posy;
      val dz: double = bodyI.posz - bodyJ.posz;
      var d2: double = dx*dx + dy*dy + dz*dz;
      //updates my bodies' velocity
      if (d2 != 0.0)
```java
var mag: double = dt/ (d2*Math.sqrt(d2));
bodyI.velx -= dx* bodyJ.mass * mag;
bodyJ.velx += dx* bodyI.mass * mag;
bodyI.vely -= dy* bodyJ.mass * mag;
bodyJ.vely += dy* bodyI.mass * mag;
bodyI.velz -= dz* bodyJ.mass * mag;
bodyJ.velz += dz* bodyI.mass * mag;

val e2 = energy2(bodyI, bodyJ, d2);
energyThisPlace+= e2;
}

//if there is only one place updates the bodies positions
if (Place.MAX_PLACES == 1){
    for (i in 0..(myBodies.size-1)) {
        myBodies(i).posx += dt* myBodies(i).velx;
        myBodies(i).posy += dt* myBodies(i).vely;
        myBodies(i).posz += dt* myBodies(i).velz;
    }
}

//calculates new target(to write) and source(to read from)
var target : Place = nextPlace.next();
var source : Place = here.prev();

/* The while loop will terminate when each place has received bodies from all other places and the source place to read from is themselves */
while (source != here) {
    if (target != here) {
        // sending toSent to the next target place
        val targetPlace = target;
        @Uncounted at(targetPlace) async {
            atomic {
                otherBodies(targetPlace.id)(pl) = toSent;
            }
        }
    }
    // waits on receipt of a set of bodies from source place
    when(otherBodies(here.id)(source.id) != null);
    //calculates all interactions with otherBodies at other place
    val other = otherBodies(here.id)(source.id);
    for (j in 0..(other.size-1)) {
        val bodyJ= other(j);
        for (i in 0..(myBodies.size-1)) {
            val bodyI = myBodies(i);
            val dx: double = bodyI.posx - bodyJ.posx;
            val dy: double = bodyI.posy - bodyJ.posy;
            val dz: double = bodyI.posz - bodyJ.posz;
            var d2: double = dx*dx + dy*dy + dz*dz;
            //updates myBodies' velocities
            if (d2 != 0.0 ) {
```
var mag: double = dt/(d2*Math.sqrt(d2));
bodyI.velx -= dx* bodyJ.mass * mag;
val e3 = energy2(bodyI, bodyJ, d2);
energyThisPlace += e3/2;
}
}

//updates my bodies' positions
for (i in 0..(myBodies.size-1)) {
    myBodies(i).posx += dt* myBodies(i).velx;
    myBodies(i).posy += dt* myBodies(i).vely;
    myBodies(i).posz += dt* myBodies(i).velz;
}

//calculates the next target and source place
target = target.next();
source = source.prev();

//offer energyThisPlace; //contributes the energy of the place to the total energy

}//after each

//printing function
public def printBodies(){
    for(pl in bodies) at(bodies.dist(pl)) {
        val bodiesHere = bodies(pl);
        for (i in 0..(bodiesHere.size-1)) {
            val body = bodiesHere(i);
            body.printBody();
        }
    }
}

//calculates the energy between two bodies residing in teh same place
public def energy1(b:Body):double {
    return 0.5*b.mass * (b.velx*b.velx + b.vely*b.vely + b.velz*b.velz);
}

//calculates the energy between two bodies from different places
public def energy2(b1:Body, b2:Body, d:double):double {
    return (b1.mass*b2.mass)/Math.sqrt(d);
}

/* module that reduces the energy values and sums the * energy from all places */
static struct mySumReducer implements Reducible[Double] {
    public def zero() = 0.0;
    public operator this(a:Double, b:Double) = (a + b);
}

Body.x10

/*
 * Implements the class Body, representing each body in the universe
 * Author: Konstantina Panagiotopoulou 2013
 */
import x10.util.Random;
import x10.io.Printer;

public class Body {
    public var mass: double; // body's mass
    public var posx : double; // body's position on x-axis
    public var posy : double; // body's position on y-axis
    public var posz : double; // body's position on z-axis
    public var velx : double; // body's velocity on x-axis
    public var vely : double; // body's velocity on y-axis
    public var velz : double; // body's velocity on z-axis

    // default constructor
    public def this() {
        this.mass = 1.1;
        this.posx = 1.1;
        this.posy = 1.1;
        this.posz = 1.1;
        this.velx = 1.1;
        this.vely = 1.1;
        this.velz = 1.1;
    }

    // constructor
    public def this(mass: double, posx: double, posy: double, posz: double,
                    velx: double, vely: double, velz : double) {
        this.mass = mass;
        this.posx = posx;
        this.posy = posy;
        this.posz = posz;
        this.velx = velx;
        this.vely = vely;
        this.velz = velz;
    }

    // helper functions
public def getMass(): double {return mass;

public def setMass(mass: double){this.mass=mass;

public def getPosX(): double { return posx;}
public def setPosX(posx: double) {this.posx=posx;

public def getPosY(): double { return posy;}
public def setPosY(posy: double) {this.posy=posy;

public def getPosZ(): double { return posz;}
public def setPosZ(posz: double) {this.posz=posz;

public def getVelX(): double { return velx;}
public def setVelX(velx: double) {this.velx=velx;

public def getVelY(): double { return vely;}
public def setVelY(vely: double) {this.vely=vely;

public def getVelZ(): double { return velz;}
public def setVelZ(velz: double) {this.velz=velz;

//print function
public def printBody(){
		Console.OUT.println("mass: "+this.getMass()+"|| X: "+this.getPosX()+"|| Y: "+this.getPosY()+"|| Z: "+this.getPosZ()+"| velx: "+this.getVelX()+"|| vely: "+this.getVelY()+"|| velz: "+this.getVelZ()+" ---PLACE: "+here.id); //"|| force: "+this.getForce());
	}

public def equals(b: Body):Boolean { // checks if two bodies are equal
	if (this.mass == b.getMass() && this.posx==b.getPosX() && this.posy==b.getPosY() && this.posz==b.getPosZ())
	{ return true; }
	else {return false;}
}
} //class Body

myRandom.x10

/*
 * Implements a random generator of double numbers
 * Author: Konstantina Panagiotopoulou 2013
*/
import x10.util.Random;
import x10.io.Printer;

public class myRandom {
    var seed: Int;
    var x: Double;

    def this(seed:Int){
        this.x = seed;
    }

    public def next(): Double{
        x = (0.456*x + 0.34) % 65386;
        return x;
    }
} //class myRandom
3. UPC

First.c

/*
 * UPC program which implements the calculation of the interactions and of teh energy
 * Also it instantiates a struct Body and calls teh main() function
 *
 * Author: Konstantina Panagiotopoulou 2013
 */

#include <upc.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <upc_relaxed.h>
#include <sys/types.h>
#include <time.h>

#define numBodies 1600*THREADS
#define chunk 1600

struct Body {
    double posx, posy, posz;
    double velx, vely, velz;
    double mass;
};

shared [*] struct Body bodies[numBodies]; //chunk; //shared array -- block dist

shared [*] struct Body otherBodies[THREADS][chunk];

double myRandom(int seed) {
    double x = seed;
    return x;
}

double next(double x) {
    x = fmod(0.456*x + 0.34 ,65386.0);
    return x;
}

void init(){
    int z = 2001;
    for (int i=0; i<numBodies; i++){
        double r = myRandom(z);
shared struct Body * b = (shared struct Body *)[i]

b->mass = next(r);
b->posx = next(b->mass);
b->posy = next(b->posx);
b->posz = next(b->posy);
b->velx = next(b->posz);
b->vely = next(b->velx);
b->velz = next(b->vely);
z+=2;

} //energy functions

double energy1(struct Body *b){
    return 0.5*b->mass * (b->velx*b->velx + b->vely*b->vely + b->velz*b->velz);
}

double energy2(struct Body *b1, struct Body *b2, double d){
    return (b1->mass*b2->mass)/sqrt(d);
}

double advance(double dt, int verbose, int debug){
    double energyThisPlace = 0.0;
    upc_forall(int i=0; i<THREADS; i++; i)
    {
        int zz=0;
        struct Body myBodies[chunk];
        struct Body toSent[chunk];

        for (int j=0; j<chunk; j++)
        {
            myBodies[j] = bodies[MYTHREAD*chunk+j];
        }

        //upc_barrier;
        if (verbose==1){
            if (MYTHREAD==0) {printf("\n========MY INITIAL-BODIES======== ");}
            for (int j=0; j<chunk; j++){
                struct Body * b = &myBodies[j];
                printf("\nBODY: mass: %f || X: %f || Y: %f || Z: %f || velx: %f || vely: %f || velz: %f || velx: %f || vely: %f || velz, b->mass, b->posx, b->posy, b->posz, b->velx, b->vely, b->velz, MYTHREAD); 
            }
        }
        upc_barrier;
        //copy myBodies to toSent
        for (int j=0; j<chunk; j++){
            toSent[j] = myBodies[j];
        }
    }
before starting computation, send my bodies (toSent) to
to next THREAD ", me, next);}
        //upc_barrier;
        
        if (next!= me ){
            for (int j=0; j<chunk; j++) {
                otherBodies[next][j] = toSent[j];
            }
            //if (debug==1){printf("ekana antigrafh ---eimai
to thread: %d \n", me);}
        }
        //upc_barrier;
        /*if (debug==1){
            printf("\n\nfor (int j=0;j<chunk;j++){
                struct Body *b = &(myBodies[j]);
                printf("--------MYBODIES-------I am : %d
and myBody in position %d is: mass: %f \n",MYTHREAD, j, b->mass);
            }
            printf("\n\nfor (int j=0;j<chunk;j++){
                shared struct Body * ob = &(otherBodies[me][j]);
                printf("--------OTHER== I am: %d and I have in myOtherBodies in pos: %d the body : mass: %f || X: %f || Y: %f || Z: %f
velx: %f || vely: %f || velz: %f --\n",MYTHREAD, j, ob->mass, ob->posx, ob->posy, ob->posz, ob->velx, ob->vely, ob->velz);
        }
    }*/

    //calculate in myBodies
    //if (debug==1){printf("------KANW TO CALCULATION ME
MYBODIES, place: %d \n ", me);}  

    for (int i=0; i<chunk; i++){
        struct Body * bI = &(myBodies[i]);
        //energy
        double en1 = energy1(bI);
        energyThisPlace+=en1;

        for (int j=i+1; j<chunk; j++){
            struct Body * bJ = &(myBodies[j]);
double dx = bI->posx - bJ->posx;
double dy = bI->posy - bJ->posy;
double dz = bI->posz - bJ->posz;

double distance = sqrt(dx * dx + dy * dy + dz * dz);

if(distance != 0.0 ){
    double mag = dt / (distance * distance);
    bI->velx -= dx * bJ->mass * mag;
    bI->vely -= dy * bJ->mass * mag;
    bI->velz -= dz * bJ->mass * mag;
    bJ->velx += dx * bI->mass * mag;
    bJ->vely += dy * bI->mass * mag;
    bJ->velz += dz * bI->mass * mag;

    if (THREADS ==1){
        double e2 = energy2(bI, bJ, distance);
        energyThisPlace+= e2;
    }
}

//if (debug==1){printf("-----TELEIWSA TO CALCULATION ME MYBODIES, place: %d \n", me);}
int source = me - 1;

//if (debug==1){printf("I am %d --- my target is: %d and my source is: %d\n ", me, target, source); }

if (target == THREADS) {target = 0;}
if (source < 0) {source = THREADS - 1;}

//if (debug==1){printf("I am %d --- my NEW target is: %d and my NEW source is: %d\n ", me, target, source); }

//upc_barrier;
while (source != me) {
    if (target != me) {
        // send myBodies (toSent) to the next target place
        for (int j = 0; j < chunk; j++) {
            otherBodies[target][j] = toSent[j];
        }
        //if (debug==1){printf("copied -- I am thread: %d \n", me);}
    }
}

// wait on receipt of a set of bodies from other place
shared struct Body *ot = (shared struct Body *) & (otherBodies[me][chunk - 1]);

if (ot == NULL) {
    while (& (otherBodies[me][chunk - 1]) == NULL) {
    }
}

//upc_barrier;
//struct Body *other = otherBodies(me);

//all interactions with otherBodies at other place
//val other = otherBodies(here.id)(source.id);
for (int j = 0; j < chunk; j++) {
    shared struct Body *bJ = (shared struct Body *) (otherBodies[me][j]);
    //if (debug==1){printf("I am : %d and otherbody J in position %d is: mass: %f \n", MYTHREAD, j, bJ->mass);}
    for (int i = 0; i < chunk; i++) {
        struct Body *bI = (shared struct Body *) (otherBodies[me][i]);
        //if (debug==1){printf("I am : %d and MYbody J in position %d is: mass: %f \n", MYTHREAD, i, bI->mass);}
        double dx = bI->posx - bJ->posx;
        double dy = bI->posy - bJ->posy;
        double dz = bI->posz - bJ->posz;

        double distance = sqrt(dx * dx + dy * dy + dz * dz);

        //update my bodies' velocity
        if (distance != 0.0) {
            double mag = dt / (distance * distance * distance);
            bI->velx -= dx * bJ->mass * mag;
            bI->vely -= dy * bJ->mass * mag;
        }
    }
}
bI->velz = dz * bJ->mass * mag;
//energy
double e3 = energy2(bI, (struct Body *)bJ, distance);
energyThisPlace += e3/2;
}

//update my bodies' position
for (int i=0; i<chunk; i++){
    struct Body * b = &myBodies[i];
    b->posx += dt * b->velx;
    b->posy += dt * b->vely;
    b->posz += dt * b->velz;
}

//upc_barrier;
/*if (debug==1){printf("-----TELEIWSA TO UPDATE ME MYBODIES, place: %d \n", me);}*/
//upc Barrier;
/*if (debug==1){ printf("\n");
    for (int j=0;j<chunk;j++){
        struct Body * b = &(myBodies[j]);
        printf("\nBODY: mass: %f || X: %f || Y: %f || Z: %f velx: %f || vely: %f || velz: %f ---THREAD: %d \n ", b->mass, b->posx, b->posy, b->posz, b->velx, b->vely, b->velz, MYTHREAD);
    }
    printf("\n");
}*/
target = target+1;
if (target ==THREADS){ target=0;}
source = source -1;
if(source<0){source = THREADS-1;}

//upc_barrier;
if (verbose==1){
    if (MYTHREAD==0){printf("\n========MY-UPDATED-BODIES======
");}
    for (int j=0;j<chunk;j++){
        struct Body * b = &(myBodies[j]);
        printf("\nBODY: mass: %f || X: %f || Y: %f || Z: %f velx: %f || vely: %f || velz: %f ---THREAD: %d ", b->mass, b->posx, b->posy, b->posz, b->velx, b->vely, b->velz, MYTHREAD);
    }
}

//upcforall
return energyThisPlace;
//advance
int main(int argc, char ** argv)
{
    int numIterations = atoi(argv[1]); //numIterations
    //printf("BODIES : %d\n", numBodies);

    int verbose = atoi(argv[2]);
    int debug = 0;
    double dt = 0.1;
    double totalEnergy = 0.0;
    clock_t begin, end;
    double time_spent;
    init();

    if (MYTHREAD==0){
        begin = clock();
        //printf("start - %f", (double) begin);
    }
    for(int i = 0; i<numIterations; i++){
        totalEnergy+= advance(dt, verbose, debug);
    }
    if (MYTHREAD==0){
        end = clock();
        time_spent = (double)(end - begin)/CLOCKS_PER_SEC;
        //printf("end - %f", (double) end);
        printf("\n Total time: %f seconds", time_spent ); // (int)
t2-t1);
        printf("\n Total energy: %f", totalEnergy );
        printf("\n");
    }
    return 0;
}

// print
int print(){
    upc_forall(int i=0; i<numBodies; i++; &bodies[i]){
        shared struct Body * bb = (shared struct Body *)(&bodies[i]);
        printf("\nmass: %f | X: %f | Y: %f | Z: %f velx: %f || vely: %f || velz: %f --from : %d\n", bb->mass, bb->posx, bb->posy, bb->posz, bb->velx, bb->vely, bb->velz, MYTHREAD);
    }
    return 0;
}
Additional Graphs

Chapel

Relative speedup

![Graph showing relative speedup for Chapel](image)

Figure 0. A Chapel - Absolute speedup graph

X10

Relative speedup
Figure 0. B X10 relative speedup graph

Larger Inputs

Figure 0. C X10 runtimes for larger inputs

UPC
Relative speedup

Figure 0.D UPC relative speedup graph