A Comparison of Parallel Languages for the Implementation Of The Kth Selection Algorithm

Neha Mittal, Roopa Kannan, Keshava P Subramanya
{nehamittal, roopa.kannan, keshava}@cs.ucsb.edu

Abstract

In this paper we present an evaluation of four contemporary parallel languages: Unified Parallel C (UPC), Message Passing Interface (MPI), STAR-P and Titanium for the Kth selection problem[1]. UPC and MPI are programming languages based on ANSI C standard whereas Titanium is based on Java and STAR-P is based on Matlab.

We consider different parameters for evaluating these languages for the current problem ranging from ease of use to abstraction to performance. We deconstruct each parallel language into its basic components, show examples, make a detailed analysis, compare them, and finally draw some conclusions.

We find MPI to be the most easy to use with the best documentation around and also to debug. The performance of MPI ,UPC and STAR-P are nearly comparable with some performing better under certain conditions.

Categories and Subject Descriptors
D.1.3 [Concurrent Programming]: Parallel programming
D.3.3 [Language Constructs and Features]: Dynamic storage management

General Terms
Performance, Experimentation, Evaluation, Parallelism

Keywords

UPC, MPI, Star P, Titanium Kth Selection problem,

1. Introduction

With the advent of multi-core systems, the interest in parallelism and parallel languages is higher now than ever before. A plethora of parallel languages are out there and it could seem to be a daunting task for one to choose between these languages to solve parallel problems. We make a humble attempt to compare four popular contemporary parallel languages, namely, UPC, MPI, Star P and Titanium. These four languages vary widely in their programming models and memory abstraction.

We start out with a problem that we consider as a representative of a typical problem that could exploit most of the abstractions of most of these above specified languages and can thus, keep our evaluation fair and reasonable. To achive this objective, we choose the Kth Selection Problem which is the core of the Parallel Sorting (PSort) algorithm. This PSort is in turn the core of many day-to-day parallel problems.

We consider the following parameters to assess various facets of above specified parallel languages based on our experiences:

- Ease of installation and deployment
- Ease of understanding and availability of help
- Ease of creation of parallel data structures
- Ease of error detection and debugging

In chapter 2, we describe the parallel languages being considered. Chapter 3 discusses the algorithm for the Single Selection problem. This is followed by a detailed experimental setup in the chapter 4.Chapter 5 presents the various parameters used to compare the languages. Finally, we showcase the performance results and

perform a comparative analysis of parallel languages being considered.

2. Description of Parallel Languages

Here we describe the four parallel languages being considered followed by some examples to illustrate their key features.

2.1. Unified Parallel C

As described by the Unified Parallel C (UPC) community[2], it is a parallel extension of the C programming language designed for high-performance computing on large-scale parallel machines, including those with a common global address space (SMP and NUMA) and those with distributed memory (e.g. clusters). There exists a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. UPC uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time, typically with a single thread of execution per processor.

In this programming paradigm, every thread runs the same program but keeps its own private local data [4]. Each thread has a unique integer identity expressed as the MYTHREAD variable. The THREADS variable represents the total number of threads used by the application. These two variables are predefined identifiers. In addition to each thread's private address space, UPC provides a shared memory area to facilitate communication among threads, and one can declare a shared object by specifying the shared type qualifier. While a private object may only be accessed by its owner thread, all threads can read or write to data in the shared address space. Because the shared memory space is logically divided among all threads, from a thread's perspective the shared space can be further divided into a local shared memory and remote one. Data located in a thread's local shared space are said to have *affinity* with the thread, and compilers can utilize this affinity information to exploit data locality in applications to reduce communication overhead.

Pointers in UPC can be classified based on the locations of the pointers and of the objects to which they point [7]. Accesses to the private area behave identically to regular C pointer operations, while accesses to shared data are made through a special pointer-to-shared construct. The speed of local shared memory accesses will be lower than that of private accesses due to the extra overhead of determining affinity, and remote accesses in turn are typically significantly slower because of the network overhead. There are three different kinds of UPC pointers: private pointers pointing to objects in the thread's own private space, private pointers pointing to the shared address space, and pointers living in shared space that also point to shared objects.

Data Distribution. UPC gives the user direct control over data placement through local memory allocation and distributed arrays [2]. When declaring a shared array, programmers can specify a block size in addition to the dimension and element type. The system uses this value to distribute the array elements block by block in a round-robin fashion over all threads. For example, a declaration of *shared [4] int array[10]* means that the compiler should allocate the first four elements of the array on thread 0, the next two on thread 1, and so on.
If the block size is omitted the value defaults to one (cyclic layout), while a layout of [ ] or [0] indicates indefinite block size, i.e., that the entire array should be allocated on a single thread

.

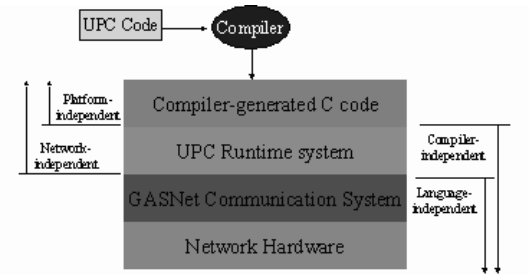

Figure 1: High level system diagram [16]

Figure 1 shows the high-level system diagram for a UPC application compiled using the Berkeley UPC compiler [16]. This is the UPC compiler being used during the entire course of experimentation. The generated C code runs on top of the UPC runtime system, which provides platform independence and implements language-specific features such as shared memory allocation and shared pointer manipulation. The runtime system implements remote operations by calling the GASNet communication interface, which provides hardware-independent lightweight, uniform networking primitives.

2.2. MPI

Message Passing Interface (MPI) [8] is a specification of message passing libraries. The goal of MPI is to provide a widely used standard for writing message passing programs. The interface attempts to be practical, portable, efficient and flexible. Interface specifications have been defined for C/C++ and Fortran programs.

Programming Model: MPI lends itself to virtually any distributed memory parallel programming model. In addition, MPI is commonly used to implement some shared memory models, such as Data Parallel, on distributed memory architectures. All parallelism is explicit in MPI. The programmer is responsible for correctly identifying parallelism and implementing parallel algorithms using MPI constructs. The number of tasks dedicated to run a parallel program is static. New tasks can not be dynamically spawned during run time.

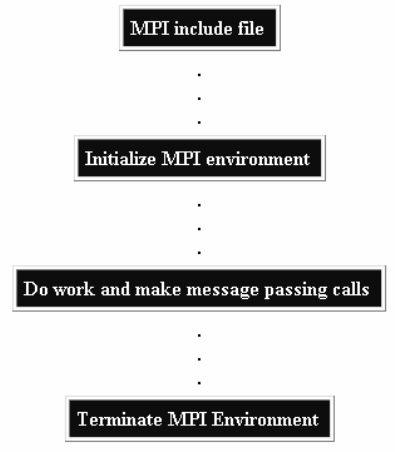

Figure2: General MPI Program Structure[10]

Communicators and Groups: MPI uses objects called communicators and groups to define which collection of processes may communicate with each other. Most MPI routines require you to specify a communicator as an argument. MPI_COMM_WORLD is a predefined communicator that includes all of the MPI processes [10].

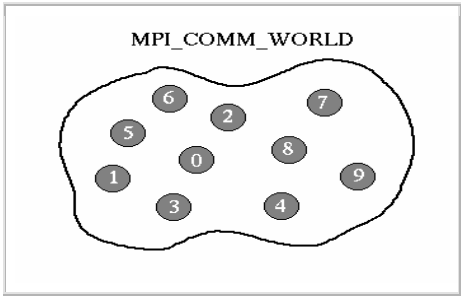

Figure3: MPI_COMM_WORLD[10]

Within a communicator, every process has its own unique, integer identifier assigned by the system when the process initializes. This unique integer is defined as Rank. It is sometimes also called a "process ID". Ranks are contiguous and begin at zero.

It is used to specify the source and destination of messages, often used conditionally by the application to control program execution [10].

MPI environment management routines are used for an assortment of purposes, such as initializing and terminating the MPI environment, querying the environment and identity, etc. Some examples of MPI environment routines are MPI_Init, MPI_Comm_size etc.

There are routines for point to point communication and collective communication routines [10]. MPI point-to-point operations typically involve message passing between two, and only two, different MPI tasks. One task is performing a send operation and the other task is performing a matching receive operation.

Collective communication must involve **all** processes in the scope of a communicator. All processes are by default, members in the communicator MPI_COMM_WORLD [10].

It is the programmer's responsibility to ensure that all processes within a communicator participate in any collective operations.

2.3 Titanium

Titanium developers describe it is an explicitly parallel dialect of Java developed at UC Berkeley to support high-performance scientific computing on large-scale multiprocessors, including massively parallel supercomputers and distributed-memory clusters with one or more processors per node [11]. Other goals of Titanium include [12]:

- **Safety:** It has two meanings in Titanium. One is the ability to detect errors statically. For instance, the Titanium compiler can ensure that all processes will execute the correct sequence of global synchronizations. The other is the ability to detect and report run-time errors, such as out-of-bound indices, accurately. Both forms of safety facilitate program development; but, not less importantly, they enable more precise analysis and more effective optimizations.

- **Expressiveness:** With built-in features such as true multi-dimensional arrays and iterators, points and index sets as first-class values, and references that span processor boundaries, Titanium is far more expressive than most languages with comparable performance.

Titanium is based on a parallel SPMD (for Single Program, Multiple Data) model of computation. It provides a global memory space abstraction (similar to UPC) whereby all data has a user-controllable processor affinity, but parallel processes may directly reference each other's memory to read and write values or arrange for bulk data transfers. A specific portability result is that Titanium programs can run unmodified on uniprocessors, shared memory machines and distributed memory machines. Performance tuning may be necessary to arrange an application's data structures for distributed memory, but the functional portability allows for development on shared memory machines and uniprocessors [11].

Titanium is essentially a superset of Java 1.4 and inherits all the expressiveness, usability and safety properties of that language [12]. Titanium augments Java's safety features by providing checked synchronization that prevents a certain classes of synchronization bugs. To support complex data structures, it uses the object-oriented class mechanism of Java along with the global address space to allow for large shared structures.

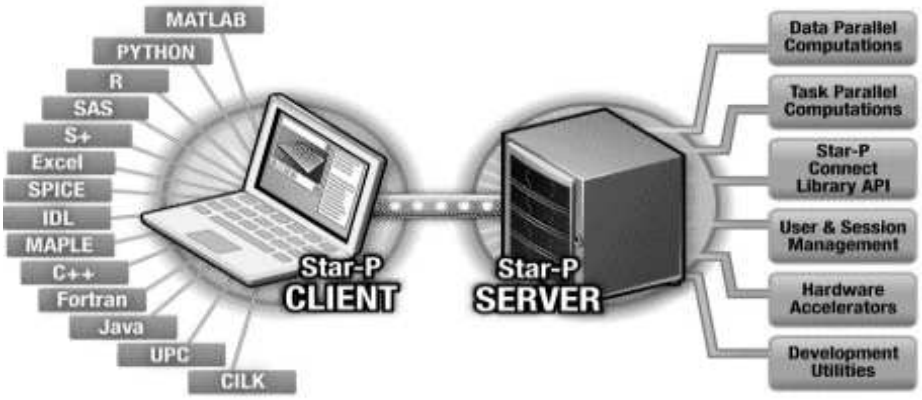

Figure 4: The Star-P open software platform delivers revolutionary results to scientists, engineers and analysts by enabling them to transparently use high performance computing resources, using familiar desktop tools [15].

2.4 Star-P

Choy et al, in their paper on STAR-P [14] describe Star-P as an interactive parallel scientific computing environment which aims to make parallel programming more accessible. Star-P borrows ideas from Matlab*P, but is a new development. Currently only a Matlab interface for Star-P is available, but it is not limited to being a parallel Matlab. It combines all four parallel Matlab approaches in one environment: embarrassingly parallel, message passing, backend support and compilation. It also integrates several parallel numerical libraries into one single easy-to-use piece of software. The focus of Star-P is to improve user productivity in parallel programming. We believe that Star-P can dramatically reduce the difficulty of programming parallel computers by reducing the time needed for development and debugging.

To achieve productivity, it is important that the user interface is intuitive to the user. For example, a large class of scientific users are already familiar with the Matlab language. So it is beneficial to use Matlab as a parallel programming language. Additions to the language are minimal in keeping with the philosophy to avoid re-learning. Also, STAR-P does not distinguish between serial data and parallel data.

```
C = op(A,B)
print(C)
```

C will be the same whether A and B are distributed or not. This will allow the same piece of code to run sequentially (when all the arguments are serial) or in parallel (when at least one of the arguments is distributed).

Leveraging both fine and coarse-grained parallelization is necessary in the vast majority of production-level HPC applications [15]. Star-P enables users to work in both a global format, and a distributed format, and to interoperate between the two.
Star-P's fine-grained parallelization enables algorithms requiring large-scale memory access and inter-processor communication, such as those found in matrix manipulation and signal processing applications. Star-P's coarse-grained mode is ideally suited for parallelization of algorithms often called "embarrassingly parallel," where computations can be naturally broken up into largely independent processes such as Monte Carlo simulation, or parallelization of FOR loops.

The Star-P interactive parallel computing platform helps desktop tool vendors leverage High performance computing (HPC) without having to solve the significant challenges associated with parallel programming and supporting

multiple HPC platforms [15]. Star-P eliminates user HPC programming and delivers interactive performance by automatically

- Sending computations to the HPC
- Splitting up the work across multiple Servers
- Providing access to world-class parallel computing libraries
- Managing inter-processor communication
- Managing the flow and memory storage of large data sets

3. Kth Single Selection Problem

The algorithm that we have implemented is very closely modeled on Saukas et al[1]. The Kth selection problem is a problem of determining the kth smallest element in an unsorted distributed array of arbitrary length. Consider a set X of n elements Given an integer k, where $1 \leq k \leq n$, the selection problem is to obtain an element x of X such that rank(x,X) = k^{+}. When k=1 and k=n, we have the special cases of determining the minimum and maximum of X, respectively. When k = n/2, we have the important case of finding the median of X. The selection problem can be solved by sequential deterministic algorithm is linear time.

This algorithm is based on successive partitioning of the input data to reduce the total amount of input elements from O(n) to O(n/p). These remaining elements can then be processed sequentially in a single processor in linear O(n/p) time.

The following partitioning strategy is used. Choose an element M in set A with the following property: $n/4 \leq rank(M,A) \leq 3n/4$.Partition A into three subsets A_l, A_e, A_g, respectively with elements less than, equal to and greater than M. We have $| A_l | \leq 3n/4$ and $| A_g | \leq 3n/4$.

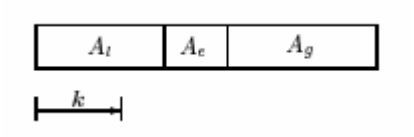

Fig. 5. Position of K relative to the sizes of A_l, A_e, A_g

Based on the value of k relative to the sizes of the three subsets A_l, A_e, A_g, we either find the kth element, or continue the selection with a smaller subset. If k corresponds to the first partition A_l, then we continue the selection using subset A_l and discard the rest (see Fig. 5); the new problem size will therefore be reduced by at least 1/4. If k corresponds to the middle partition A_e , then we have found the answer: the kth element is M . If k corresponds to the last partition A_g, then we continue the selection using subset A_g to select the element of rank k (A_l + A_e); again the new problem size will be reduced by at least 1/4.

Let n_i be the number of remaining elements in processor i at each round and $N = \sum_{1 \text{ to } p} n_i$. We consider m_i as the median of the remaining elements in processor i. The weights w_i are given by $w_i = n_i / N$. Therefore using the weighted median we not only consider the values of the medians in each block, but also the amount of remaining elements in each block. This will allow the subsequent rounds to deal with different number of elements in each processor and nevertheless guarantees the reduction by at least 1/4 of the total number of remaining elements at each round.

The actual algorithm is as follows [1]:

3.1 Algorithm *:*

Single Selection Algorithm for obtaining the k_{th} smallest element, assuming p processors each with local memory of size O(n/p).

Input: *Set A of n elements distributed among the p processors, each processor i with n_i = O(n/p) elements, and an integer k, $1 \leq k \leq n$.*

Output: *An element a i of A such that rank (a_i , A) = k.*

(1) Set N: = n

(2) Repeat until $N \leq n / (cp)$
(2.1) Each processor i computes the median m_i of its n_i elements
(2.2) Each processor i sends m_i and n_i to processor 1
(2.3) Processor 1 computes the weighted median M
(2.4) Processor 1 broadcasts M to all other processors
(2.5) Each processor i computes l_i , e_i , g_i , respectively the numbers of its local elements less than, equal to, or greater than M
(2.6) Each processor i sends $l_i \leq e_i \leq g_i$ to processor 1
(2.7) Processor 1 computes $L = \sum_{1 \text{ to } p} 1i$, $E = \sum_{1 \text{ to } p} e_i$, $G = \sum_{1 \text{ to } p} g_i$ respectively the total numbers of elements less than, equal to, or greater than M (2.8) Processor 1 broadcasts L; E; G to all other processors
(2.9) One of the following:
if $L < k \leq L + E$ then return solution M and stop
if $k \leq L$ then each processor i discards all but those elements less than M and set N := L
if $k > L + E$ then each processor i discards all but those elements greater than M and set N := G and k := k - (L +E)

(3) All the remaining N elements are sent to processor 1

(4) Processor 1 solves the remaining problem sequentially

| End of Algorithm |

3.2 Weighted Median

Given p distinct elements m_1, m_2, m_3…. m_p with corresponding positive weights w_1,w_2,… w_p such that $\sum_{1 \text{ to } p} w_i = 1$ the weighted median is the element m k that satisfies [1]

$$\sum_{i,m_i<m_k} w_i \leq \frac{1}{2} \text{ and } \sum_{i,m_i>m_k} w_i \leq \frac{1}{2}.$$

4. Experimental Setup

Our experimental setup consists of a 8-way Dual-Core AMD Opteron (tm) Processor 8214 making a total of 16 processesing units and capable of simultaneously executing 16 parallel threads. The Operating System running is x86-64 GNU/Linux and the kernel version is 2.6.18-8.1.4.el5 with SMP support enabled.

5. Language Evaluation Parameters

We adopt a multi-faceted approach in our assessment of these parallel languages. In particular, we look at the following parameters of each of the languages and analyze the languages based on our experiences with them.

- Ease of installation and deployment
- Ease of understanding and availability of help
- Ease of creation of parallel data structures

- Ease of error detection and debugging

5.1 Ease of installation and deployment

The installation of UPC was pretty much straightforward since the install file had detailed explanation of the steps.
To keep the evaluations fair we wished to link C++ standard template library (STL) code to our UPC code. This required us to use C++ with in the pure C nature of UPC.
We were able to achieve this by creating an extern C function and linking the prototype to both C and C++ compilation units and using a special compiler directive "*-link-with=g++*".

Here are excerpts from the code that achieve this.

Figure 6 illustrates an example header file which should be imported by both C and C++ compiler

```
#ifdef __cplusplus
    extern "C" {
#endif
     int myFunc();
#ifdef __cplusplus
      }
 #endif
```

Figure 6

Titanium- The installation was also smooth albeit lengthy. There were more than one components to be integrated which made the task a little difficult. The absence of smooth integration with an eclipse like IDE proves to be an impediment for seasoned programmers well versed with java programming using eclipse or netbeans. Furthermore the language varies significantly from Java in that several new syntactic constructs have been added which requires a learning curve. We did manage, with some amount of tweaking to be able to run the eclipse IDE linking with the new classes of Titanium.

The MPI and Star-P are already installed and available on the experimental setup.

5.2 Ease of Understanding and availability of help

The MPI programming model came naturally to us and visualization of data movement using message passing primitives was very intuitive. Furthermore, clear crisp and abundant documentation of MPI made programming for MPI a breeze. The use of high-level routines for data reorganization among processors with efficient under-the-hood implementation made MPI a lean mean parallel machine. Also, smooth integration with C++ enabled the use of STL without the use of any tweaks and thus helped us to use several state-of-the-art algorithmic implementations and thus cut down the application development time.

The UPC language, on the other hand, was relatively hard to work with. The non-availability of bucketed n to n data transfer seemed to cut down the utility of UPC in problems that require exactly this kind of communication. Also, the official documentation, in general, seems to be very terse.

The primary help of Titanium came from Language reference manual, which describes the full syntax and semantics of the Titanium language. This documentation contained fewer code snippets and sample programs than one would have liked. As far as understanding, it was a breeze because it came bundled with the familiarity and comfort of Java.

Star-P clearly follows MATLAB in that the availability of good help and documentation is in abundance. However we got access to these manuals from private machines, we are unsure if they are available for general public yet.
It took us significantly less time to develop a working prototype in Star-P than any of the other languages owing to good availability of help and high level commands.

5.3 Ease of creation of parallel data structures

We used upc_all_alloc primitive to create a distributed dense array which is shared across all threads and is compatible with the following declaration.

*shared [nbytes] char[nblocks*nbytes]*

Here we were faced with the problem of specifying the value of nbytes as a compile time constant and therefore we couldn't use a runtime variable like THREADS (the total number of threads used in the application). This poses a problem when we need to divide an array of arbitrary length nearly equally among 'n' available processors and have contiguous blocks of n/nproc have affinity to each processor.

Finding the right distributed data structures was the hardest thing we did in Titanium. The divergence of array (grid) in Titanium from Java as first class citizens, and the various nuances of the new syntax made it harder than necessary to create dynamic distributed data structures.

Star-P was the most shining penny in the fountain as far as creation of global data structures was concerned. The abstraction hid away the distribution of arrays and gave a monolithic view to the programmer and hence was very easy to deal with.

5.4 Ease of Error detection and Debugging

In UPC due to the barrier synchronization primitive and implicit receive at the barrier, debugging seems to be not very convenient. On the other hand explicit message passing primitives of the MPI, make debugging simpler at the expense of extra lines of code. In both these languages the error output was similar to the error output in regular C and C++ programming and hence was relatively easy to handle.

Titanium on the other hand displayed compilation error messages very distinct from Java and hence took some getting used to.

Our experience with Star-P was that due to its command line/ high level nature error detection and debugging was some what hard.

6. Performance Benchmarking

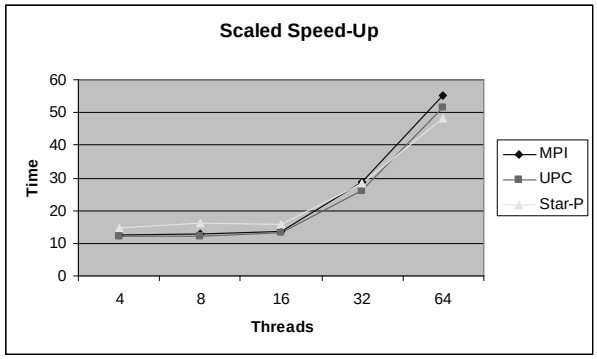

Figure 7: Scaled Speed-Up

Scaled speedup means that you increase the size of the problem at the same rate that you increase the number of processors, so the number of vector elements per processor remains constant.

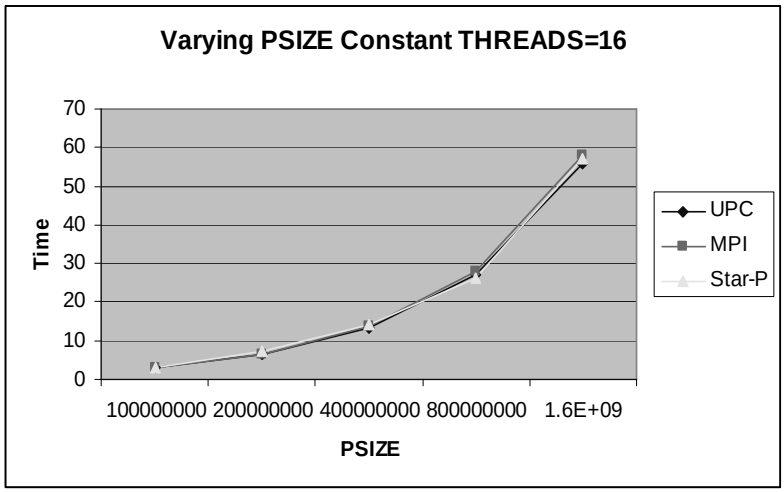

Figure 8: Varying PSIZE Constant THREADS

Figure 7 shows the scaled speed-up graph of MPI, UPC and Star-P. As it is clear from the graph, we see that the scaled speed up of the three languages is almost comparable.

Figure 8 represents the performance under increasing PSIZE and fixed number of THREADS. We see that the time taken is strictly linearly increasing as expected.

	UPC	MPI	Star-P	Titanium
Ease of installation and deployment	Good	-	-	Average
Ease of Understanding and availability of help	Average	V.Good	Good	Average
Ease of creation of parallel data structures	Average	-	Easy	Hard
Ease of error detection and debugging	Average	Easy	Average	Hard

Figure 9: Evaluation of the 4 languages

7. Conclusion and Future Work

Figure 9 summarizes our experiences with the four parallel programming languages. In our opinion, but for the initial learning curve we feel that Titanium has the potential of becoming the de-facto standard for parallel programming languages as it addresses some of the real concerns of the community. The familiarity of Java is a big plus for Titanium. Performance wise we find that Star-P,UPC and MPI are comparable and Star-P has the lowest number of lines of code.

As Future work it would be interesting to study more facets of parallel languages. We would also like to complete our Titanium program and get the performance numbers of it, so that we can compare it with the other languages.

References:

[1] E.L.G. Saukas and S.W.Song, *A note on Parallel Selection on Coarse Grained Multicomputer*

[2] Ami Marowka, *Analytic Comparison of Two Advanced C Language-Based Parallel Programming Models*, Proceedings of the ISPDC/HeteroPar'04, 2004 IEEE

[3] UPC: http://upc.lbl.gov/

[4] UPC: http://upc.gwu.edu/

[5] Sebastien Chauvin et al, UPC Manual, George Washington University

[6] UPC: http://upc.lbl.gov/docs/user/

[7] Chen, Bonachea, Duell, Husbands,Iancu, Yelick, *A performance Analysis of the Berkley UPC Compiler*

[8]MPI: http://www.llnl.gov/computing/ tutorials/mpi

[9] MPI: http://www.mpi-forum.org/docs/ mpi-11-html/mpi-report.html

[10] MPI: http://www.mhpcc.edu/training/ workshop/mpi/MAIN.html

[11] Titanium Homepage: http://titanium.cs. berkeley.edu/

[12] Kathy Yelick et al, *Titanium: A High-Performance Java Dialect*, In Concurrency-Practice and Experience, Java Special issue, 1998

[13] Katherine Yelick et al, *Parallel Languages and Compilers: Perspective from the Titnaium experience, June 7, 2006*

[14] Ron Choy, Alan Edelman, John R. Gilbert, Viral Shah, David Cheng, *Star-P: High Productivity Parallel Computing*

[15]STARP: http://www.interactivesuper Computing.com/products/

[16] GASNet: http://gasnet.cs.berkeley.edu/

[17] Weighted Median: http://books.google.com/books?id=NLngYyWFl_YC&pg=PA194&lpg=PA194&dq=cormen+weighted+median&source=web&ots=BvVnID3jDe&sig=9cSatCudd6gpydYH-0vLeAWqUQw

[18] Selection Algorithm:http://en.wikipedia.org/wiki/Selection_algorithm

[19] UPC/C++: http://upc.lbl.gov/docs/user/interoperability.html

Appendix

Mersenne Twister(MT) is a pseudorandom number generating algorithm developped by Makoto Matsumoto and Takuji Nishimura (alphabetical order) in 1996/1997. An improvement on initialization was given on 2002 Jan.
MT has the following merits:

- It is designed with consideration on the flaws of various existing generators.
- The algorithm is coded into a C-source downloadable below.
- Far longer period and far higher order of equidistribution than any other implemented generators. (It is proved that the period is 2^19937-1, and 623-dimensional equidistribution property is assured.)
- Efficient use of the memory. (The implemented C-code mt19937.c consumes only 624 words of working area.)