

# Port of a real world application to Chapel

Colin MacSweeney  
Summer Student  
Irish Centre for High End Computing

August 18, 2009

## Abstract

The aim of this project was to port a real world application to Chapel, which is a new high productivity, parallel programming language. The application to be ported was selected from a collection of applications written by users of ICHEC's High Performance Computing Resources.

## 1 Existing Parallel Technologies

### 1.1 MPI

Message Passing Interface (MPI) is a specification of a language independent API commonly used to write parallel programs. MPI was designed by a committee consisting of library writers, vendors, and application specialists. There are many different implementations of MPI including MPICH2 and OpenMPI which allow MPI programs to be executed. There are MPI bindings for many different languages but the majority of MPI programs are written using C/C++ and Fortran. MPI is a Single Process, Multiple Data (SPMD) technology in which the user writes a single program which is executed simultaneously on multiple processes. Each process is parameterized with a unique ID so the user can control the tasks performed on each process.

### 1.2 OpenMP

OpenMP is an API designed to enable portable shared-memory parallel programming. Open MP provides simple mechanisms which allows a collection of cooperating threads to be created to perform tasks. The shared-memory model means that most variables in OpenMP programs can be accessed by all threads. OpenMP consists of a set of compiler directives, library routines and environment variables which describe parallel operations in Fortran and C/C++ programs. In C++, OpenMP compiler directives all begin with '#pragma omp'. Version 3.0 was released in May 2008 and this is supported by many compilers including the latest versions of GCC and the Intel Fortran and C/C++ compilers.

## 2 Chapel

Chapel is a new, high productivity, parallel language being developed by Cray Inc. It is part of the High Productivity Computing Systems (HPCS) program which is funded by the Defense Advanced Research Projects Agency (DARPA). Other new languages included as part of this program include X10 which is being developed by IBM and Fortress which is being developed by Sun. Chapel aims to remedy some of the weaknesses of the existing, predominant parallel languages such as MPI and OpenMP. The language features a multi-resolution design to give developers maximum flexibility while programming in Chapel. This multi-resolution design means that users have the choice of using very high level, abstract programming constructs, or, if the user wants more control of the implementation, he/she may also write code at a lower level to specify exactly what should be performed. In this way Chapel aims to provide the benefits of both high level and low level languages.

In contrast to MPI, Chapel adopts a global view programming model. This means that the user writes code which describes the entire computation as a whole rather than describing the work performed by each processor. This is usually not the case for MPI, which uses a Single Process, Multiple Data (SPMD) programming model. Chapel provides high level, global-view abstractions for performing data parallelism, task parallelism and nested parallelism which aim to simplify the development of parallel programs. The current version of the Chapel compiler is Version 0.9 which was released on April 16th 2009.

## 3 Keeling Algorithm

The program which was chosen to be ported into Chapel for this project was written by Dimitri Perrin who is a user of ICHEC's HPC resources. The program generates a social network using an extended version of the algorithm which was first proposed by Keeling. The network which is created by this code will be utilised in an agent based simulation designed to investigate how a disease spreads among a population. The original code by Dimitri Perrin was written in C++ with parallelism was achieved through MPI.

In a social network graph, a node represents a person and an edge represents a relationship between two people. The algorithm takes as an input  $N$ , the number of nodes, which are then randomly distributed in a two dimensional square area of side length  $\sqrt{N}$ . This network of nodes is divided into subnetworks which are then distributed among the compute nodes which is the program is being executed on. A number of focal points are also created and distributed among the nodes in a similar manner. The number of focal points is determined by multiplying the parameter  $F$  by  $N$ . These focal points represent population centres. Each of the nodes is translated towards its nearest focal point by a certain proportion in order to increase the population density in these areas. The proportion by which nodes are moved towards their focal points is set to 0.6 by default however this could be increased or reduced to increase or decrease the resulting population's density near focal points. The algorithm generates edges between some of the nodes in each subnetwork using a probability function which is based on the Euclidean distance between 2 nodes, the height  $H$  and the Variance  $V$ . The parameters  $H$  and  $V$  are set to 1.5 and 0.5 by default however these can be configured to customise the created edges. The formula is shown below:

$$P(\text{edge}) = \min \left\{ \begin{array}{l} H \exp \frac{-d^2}{2V} \\ 1 \end{array} \right.$$

Once the edges linking nodes within each subnetwork are created, the next step of the algorithm is to link certain subnetworks based on another probability function. The probability of two subnets being linked is given by:

$$P(\text{subnet link}) = \frac{0.5}{S}$$

where S is the total number of subnetworks.

When two subnets are linked, random nodes from each subnet are linked together. The number of links generated between two connected subnetworks is defined by a constant which can be customised.

## 4 Mersenne Twister Pseudorandom Number Generator

The code written by Dimitri Perrin also makes use of a pseudorandom number generator known as the Mersenne Twister. This is a famous generator which was developed by Makoto Matsumoto and Takuji Nishimura in 1997. This pseudorandom number generator has a period of  $2^{19937-1}$  and its order of dimensional equidistribution is 623. To demonstrate that Chapel could be used for writing very low level code in addition to high level code, I decided to port this generator to Chapel also.

The algorithm is a twisted Generalised Feedback Shift Register (GFSR) which is based on the following recurrence relation:

$$x_{k+n} := x_{k+m} \oplus (x_k^u | x_{k+1}^l)A, \quad (k = 0, 1 \dots)$$

where:

$\oplus$  is bitwise XOR

$|$  is bitwise OR

n is the degree of the recurrence

$x_k^u$  is the upper bits of a word

$x_{k+1}^l$  is the lower bits of a word

## 5 Setting up the Chapel Environment

For this project, I used one of ICHEC's High Performance Computers known as 'Stokes'. Stokes is an SGI Altix ICE 8200EX featuring 320 compute nodes. Each node contains two Intel Xeon quad-core processors and 16GB of RAM and the nodes are connected using ConnectX Infiniband.

I used version 0.9 of the Chapel compiler (released on April 16th 2009) to compile the chapel code. Before building the compiler it was necessary to set the following environment variables as I was using the GASNet communication library to manage communication between processes.

```
CHPL_COMM = gasnet
CHPL_COMM_SUBSTRATE = ibv
```

The version 0.9 release of the compiler only supports GASNet as the communication mechanism however support for Open MPI has since been added to Chapel. There are also GASNet environment variables for execution which need to be set to execute the chapel program once it's compiled.

```
GASNET_SPAWNFN = S
```

This tells GASNet to spawn program instances using ssh. It is also necessary to tell GASNet which nodes to connect to. This was done in the pbs script used to launch the job with the following line which retrieves the nodes names from the PBS Nodefile and replaces the carriage returns with spaces:

```
export GASNET_SSH_SERVERS='cat $PBS_NODEFILE | uniq | tr '\r\n' ' '
```

Note that the variable name is "GASNET\_SSH\_SERVERS" and not "SSH\_SERVERS" as stated in the Chapel documentation.

## 6 Locality in Chapel

The Chapel language defines a *Locale* as a portion of the target parallel architecture that has processing and storage abilities.[1] When running the chapel program on Stokes, each node of Stokes is treated as a different locale. Each locale has its own local memory which it can access with much shorter latencies than memory on other locales. For this reason it is very important when programming in Chapel to minimize the number of non-local memory accesses as these will heavily impact on performance.

The Chapel compiler generates two executable files after compiling the Keeling program as a multi-locale program - "keeling" and "keeling\_real". "Keeling\_real" is the keeling program itself and "keeling" is the launcher program used to spawn it on multiple locales. The program can be run on four locales with the command ". /keeling -nl 4"

When declaring arrays in Chapel, they may be distributed among Locales according to a *Distribution*. The Chapel developers have defined a Block Distribution which distributes an array evenly in blocks among all locales. Custom distributions can also be defined by users by creating a subclass of the *Distribution* class. To distribute an array in chapel, a domain is first created which declares the set of indices and the distribution which will be used. The array is then declared on that distribution. The following lines show how the nodes in the social network were distributed using the block distribution provided by chapel where nbNodes is the total number of nodes.

```

const keelingDist = new Block1D(idxType = int, bbox = [0..nbNodes-1]);
var keelingD: domain(1) distributed keelingDist = [0..nbNodes-1];
var keelingArray: [keelingD] keelingPoint;

```

The first line creates the distribution, the second line creates the domain and the the third line creates the array.

## 7 Porting from C++/MPI to Chapel

As MPI uses a SPMD programming model and Chapel uses a global-view model, I decided to make small alterations to the social network program when porting it to Chapel. It would have been possible to create a Chapel version of the code which resembled the MPI version more closely however this would not have taken advantage of many of Chapel's high level constructs such as locales and forall loops which free the programmer from having to describe the computation from a single processor point of view.

The MPI implementation of the program begins by dividing the total number of nodes by the number of subnetworks and then allocating this many nodes on each subnetwork. To perform this step in Chapel only one array needed to be created which was then distributed among the locales using a block distribution. A forall loop was utilised to allocate each element of the array on its corresponding locale in parallel. The MPI code creates one subnetwork for each processor which is executing the program, however, for the Chapel version I decided to modify this and create one subnetwork per locale and specify that the number of tasks on each locale be the minimum number of cores on any locale as illustrated below.

```

const tasksPerLocale = min reduce Locales.numCores;

const keelingDist = new Block1D(idxType = int, bbox = [0..nbNodes-1],
                                tasksPerLocale = tasksPerLocale);
var keelingD: domain(1) distributed keelingDist = [0..nbNodes-1];

```

This takes greater advantage of locality in Chapel and the result is that one subnetwork is created per node when running on Stokes. This method increases the number of internal edges generated by subnetworks however this is now performed in parallel within each subnetwork and the number of subnetworks which need to be connected is also reduced. Each node is represented as an object of the class KeelingPoint which contains two fields named attribute and position.

After allocating the nodes, the next step in the algorithm is to allocate the focal points. These are distributed among the locales in exactly the same manner as the nodes. Next, each of the nodes are assigned attributes. Currently these attributes are given random values in the range 0 to 1. The original author of the MPI program intends to later change this to read the attribute values from a file. In Chapel, I was able to take advantage of a high level function provided in the Chapel Random module which allows an entire array to be filled with random numbers in parallel in a single statement.

```

fillRandom(keelingArray[i].attribute);

```

This function is aware that `keelingArray` is a distributed array and will perform the operation for each element on the locale which that element is stored on. After assigning the attributes for each node, next the position of each node is determined based on the attribute values. I used a *forall* loop in Chapel to perform this in parallel. Each iteration of the forall loop is executed on the locale which the corresponding element of the array is stored on. The *getNext* function in the `Random` module is not thread safe so it was also necessary to use a boolean sync variable to prevent multiple threads calling this at the same time. This is a special type of variable in the Chapel language which allows controlled access to a critical section.

```
forall i in keelingD {
    for j in 0..(dimension - 1) {
        lock$ = true;
        keelingArray[i].position[j] = randStream.getNext() * max * keelingArray[i].attribute[j];
        lock$;
    }
}
```

Nodes are translated towards to their nearest focal point using another parallel forall loop. Inside the forall loop, the code first determines which is the nearest focal point which is stored on the same locale as the node. It is important that only focals on the same locale are accessed in order to prevent remote memory accesses which would cause a performance drop.

```
forall i in keelingD {
    ....
    for j in (focalsPerLocale * here.id)..(focalsPerLocale * here.id) + focalsPerLocale - 1 {
        ...
    }
}
```

”`here.id`” is a predefined Chapel constant which refers to the locale that the current task is running on. The above for statement takes advantage of this identifier to iterate through the focals on the same locale.

The edges between nodes are created using another parallel forall loop. The probability function that determines which nodes to connect is contained in the *KeelingProb* function and the edges that are generated are stored in a sparse matrix. In the MPI version of the program, a class named `sparseMatrix` is used to store this data. This class contains an array which grows dynamically as more edges are added. A separate instance of this class is created for each process which stores the edges corresponding to that subnetwork.

There is a sparse array type built into Chapel which would have been ideal to represent this matrix however it does not yet support being distributed across locales. Also, Chapel does not yet support arrays of arrays where the inner arrays vary in size. This is because an array cannot be reassigned

to a new domain after it is first created. As a result, it was not possible to implement the sparse matrix in a similar manner to the MPI program. For this reason, I decided to create my own linked list class and use an array of these objects. Each linked list contains the edges for a node and the array containing the lists is distributed among the locales using a standard block distribution.

The edges connecting nodes in different subnetworks are generated afterwards and are stored in a separate array. This task is not performed in parallel and therefore the Chapel code bears close resemblance to the MPI version. The program saves the internal edges of nodes and the inter-subnetwork edges to a file named `Metakeeling.dat` before terminating.

## 8 Conclusions

The goal of this project was to investigate the differences in programming in Chapel as opposed to a currently prevalent HPC language. The project aimed to use the high level constructs which have been introduced into Chapel with the goal of increasing programming productivity. These high-level abstractions reduce the volume of code which needs to be produced, however it is necessary for the programmer to have a thorough understanding of how the compiler interprets these statements in order to write efficient and effective code. In MPI, the developer must explicitly tell the compiler when he/she wants to send or receive data to or from another process. In Chapel however, the compiler will do this transparently without giving any warning, even though it will be much slower than accessing local memory. This means that when writing code in Chapel, the programmer must be aware of the locale which data is stored on and also the locale which tasks are running on. By minimizing the number of times remote memory is accessed, maximum performance can be achieved.

A large number of Chapel's features are available in the current release (version 0.9) of the compiler however there are still many features which are not yet implemented. The 'STATUS' text file which is included with the compiler gives a detailed account of what remains to be implemented in the compiler and also known bugs. One of the most notable bugs is that there are currently memory leaks in the compiler. For this reason, the compiler does not seem ready to be used for large scale projects. Also, distributed domains and arrays are still under development, atomic statements are not supported and the language has not yet been optimized to provide fast performance. As Chapel is a language which is rapidly changing, it is very difficult for the documentation to be updated to reflect every change. This added to the challenge of programming in Chapel as the documentation did not always reflect the current state of the language.

Overall, I think the language will succeed in its goal to improve the productivity of HPC programmers. The usefulness of Chapel's higher level parallel operations and distributed data operations were clearly evident in this project.

## References

- [1] *Chapel Language Specification 0.782*. Cray Inc. 2009
- [2] *Parameterisation of Keeling's network generation algorithm* Jennifer Badham, Hussein Abbas and Rob Stocker, Artificial Life and Adaptive Robotics Laboratory, School of ITEE, Australian Defence Force Academy 8 February 2007

- [3] *Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator* Makoto Matsumoto and Takuji Nishimura Keio University