CS510 Languages and Low Level Programming: Portfolio submission, Topic 10

Due on June 3, 2016 at 11:59pm

Mark P. Jones Spring 2016

Konstantin Macarenco

Topic 10. Use practical case studies to evaluate and compare language design proposals.

With the lack of specific language for Low Level Programming, I picked two languages with Parallel Programming in mind (General C + MPI library, and Chapel - new domain specific language for parallel programming created by Cray). Even though domain is different it shows that language designed for a specific domain can drastically aid applications development. Parallel programming complexity is similar, if not greater than LLP, with many potential issues on the top of regular mistakes there are Concurrency Issues such as race conditions, and deadlocks. MPI - message passing interface is a well known standard for external parallel computing. Chapel - is build upon C/MPI is much simpler - it is a modern high level language that hides all the intricacies of Message Passing. Chapel syntax is similar to Python, with many alike features. Chapel uses C/MPI as an intermediate layer, i.e. first it compiles to C.

I Compare implementation of Jacobi-Laplace algorithm in C/MPI vs Chapel.
Jacobi-Laplace is a simple approach for solving Laplace equation with $O(n^3)$ complexity, used in many scientific applications. As $n \to \infty$ performance is greatly reduced, hence the desire to run it in parallel mode.
Laplace equation : ${\phi^{t+1}}_{i,j} = \frac{1}{4}({\phi^t}_{i+1,j} + {\phi^t}_{i-1,j} + {\phi^t}_{i,j+1} + {\phi^t}_{i,j-1}), 0 < i, j < N$, i.e. current cell in a matrix is equal to a quarter of the sum of it's neighboring cells.

External implementation requires matrix partitioning, and message passing when computing border elements.

Implementation comparison chart - next page.
Both implementations are included to the submission

- laplace-mpi.c - C/MPI
- chapel-distr.chpl - Chapel
- Makefile
- laplace-distr.c - Chapel Generated code (not to be compiled or executed).
- chpl-prep - Environment setup for running chapel appliactions
- mpi-prep - Environment setup required for running MPI applications

Building and running instructions:

The code intended to be build on linuxlab machines.

```
make
# MPI -
    source mpi-prep && mpirun -n 4 laplace_mpi  # number of remote procs must be 4 for
        this application
# Chapel
    source chpl-prep && ./laplace-distr -nl [number of remote procs] # number of
        procs can not exceed 20
```

Table 1: Languages comparison

Types	C/MPI	Chapel
Complexity	High. It is quite a challenging task to do int C/MPI, since matrix mapping to the network done manually, and all communications are manual as well. The particular implementation I use as an example, divides matrix to 4 regions, each region is then sent to a remote CPU. It would be even more challenging to implement dynamic scalable matrix partitioning.	Low. As easy as implementing sequential version, just need to specify domain mapping. Chapel sequential and MPI implementations are almost identical, with one exception - matrix needs to be mapped to the distributed cluster. This is a natively supported operation in Chapel, short and concise. Unlike C doesn't require any math, or matrix offsets calculation. Code is simple and easy to read. Chapel includes multiple flexible partitioning schemes, and also allows custom definition. The problem is mapped automatically depending on the number of the available remote CPUs.
Error Prone	No. C doesn't provide much help in error detection, or debugging, if in case of any fault, especially related to race conditions	Yes. Chapel can detect common programming error such as division by zero, out of index, etc.
Effort	High.	Low.
Code Size	Large ≈ 620 *lines* with comments	Small ≈ 60 *lines* with comments
Performance	High. Area where C shines is performance. C implementation is up to x40 times faster.	Low. At this point of time Chapel suffers from performance issues. It is a new language, that is still undergoing major development, so code produce by chapel compiler is far from being optimal. For example generated C code is $\approx 3K$ lines long, besides the size, generated code is far from being optimal.

In conclusion: this comparison clearly demonstrates how usage of domain specific language can simplify the problem and increase developing performance.