Parallel Programming Constructs and Techniques using an Embedded Flexible Language (EFL) for Python

M. Goldstein [a,b], D. Dayan [a,b], D. Berlowitz [a], O. Berlowitz [a], M. Rabin [a], M. Nagar[a], D. Soudry[a], M. Naaman[a], E. Bussani Levy[a], R. B. Yehezkael [a]

(a) Department of Computer Science, Jerusalem College of Technology, Jerusalem 91160, Israel.
(b) The Unit for Programming Teaching, The Hebrew University of Jerusalem, Jerusalem 91904, Israel.

Motivation

Multi-core CPUs are abundant and utilizing them effectively requires programmers to parallelize CPU-intensive code. Because of the *heterogeneity of parallel programming platforms* today (MPI, openMP, Python's Threads and Multiprocessing modules, etc.), there is a need for a *common approach* which will free the programmer from the platform technical intricacies.

Objectives

A major objective has been to develop a straightforward language which implements this common approach. This approach should allow flexible execution in which sequential and parallel executions produce identical *deterministic* results.

To facilitate this, we have developed EFL, a deterministic parallel programming tool.

EFL Programming Model

Well-defined *interleaving* of sequential and deterministic parallel code blocks allows the programmer to use both as appropriate:

```
... host language sequential code ...        (1)
EFL{
If (a > b) {
    x = f(a);
} else {                                     (2), (3)
    y = f(a);
}
z = g(b);
}EFL
... host language sequential code ...        (1)
```

(1) The sequential parts of the program are written in the host language.

(2) EFL syntax: C-style and host-language independent.

(3) EFL semantics:

- Deterministic, like FP, but efficient memory management is taken care by the procedural host language.
- Implicit parallelism is implemented by translating embedded EFL blocks of code into explicit parallel host-language code.

Three kinds of limitation rules enforced in order to ensure deterministic parallelization:

(a) The programmer may call "pure" functions only.

(b) In and Out variables (but not InOut variables!).

(c) Once-only assignments.

EFL Implementation

Until now, two Python implementations of an EFL pre-compiler have been developed based on:

(1) Python's Multiprocessing module

(2) The DTM/MPI4PY modules

EFL Building Blocks

EFL Basic Constructs

Assignment-block
It is a set of assignment expressions.

```
EFL{
  // Simple assignment of value
  myValue = 5;
  // Expression containing function call
  myExpr = f(5);
}EFL
```

Pif-block
This construct allows for all of the conditions of the pif to be evaluated in parallel.

```
EFL{
  pif (a == 0) {
    //something
  } elseif ((b / a) == c) {
    //possible divide by zero exception
  } else {
    //something else
  }
}EFL
```

If-block
It behaves as a sequential If statement, but it is executed in parallel relative to the rest of the EFL block.

```
EFL{
  if (cond1(someValue)) {
    // some code here
  } elseif (cond2(someValue)) {
    //other code here
  } else {
    //last code here
  }
}EFL
```

For-block
Each iteration instance is executed in parallel to the rest of the iteration instances..

```
EFL{
  for (i = 0; i < 10; i = i + 1) {
    mySeq[i] = cpuIntensiveCalc(i);
  }
}EFL
```

EFL Parallel Constructs

LogLoop-block
It allows performing an efficient parallel scan of a sequence.

```
EFL{
  result = logloop(mySeq, myFunc);
}EFL
```

MapLoop-block
Like the "map" function, it applies in parallel a function to every element of a sequence.

```
EFL{
  sqrts = maploop(mySeq, sqrt);
}EFL
```

Loop-block
All the iteration instances may be executed in parallel - the loop body contains its own local variables for each index value of the loop.

```
EFL{
loop (i = 0) :
{
    if (i < len(seqIn))
      loop(i + 1);
    seqOut[i] = cpuIntensiveFun(seqIn);
}
}EFL
```

Programming in EFL: Multiplying a Matrix by a Vector

```
def Mult(matRow,vec):
        res =[0,0,0]
        ret = 0
        print "mult" , matRow ,"X", vec
        EFL{
                for (j = 0; j < len(matRow); j = j + 1) {
                        res[j] =  matRow[j]*vec[j] ;
                }
        }EFL
        for item in res:
                ret =  ret + item
        print "res =", res, "->",ret
        return ret
```

```
def Main():
        mat = [[1,2,3],[4,5,6],[7,8,9]]
        vec = [1,2,3]
        print "Multiplying A Matrix And A Vector"
        print mat ,"X", vec
        res = [0,0,0]
        print "length of mat : ", len(mat)
        EFL{
                for (i = 0; i < len(mat); i = i + 1) {
                        res[i] = Mult(mat[i],vec);
                }
        }EFL
        print "Final result = ", res
if __name__ == '__main__':
        Main()
```

Parallel Programming Patterns in EFL

The Parallel Programming Patterns we implemented using EFL are: Fork Pattern, Master-Worker Pattern, Pipeline Pattern, Geometric Pattern and Nesting Pattern.
We present here alternative EFL codings for implementing the Master-Worker Pattern.

```
import sys
import time
def p(workerId):
        x1=range(1000)
        for i in range(10000):
                for j in range(1000):
                        num1 = j + workerId
                        x1[j] += num1
        return (workerId, sum(x1))
start_time = time.clock()
resL = range(5)
EFL{
        resL[0] = p(0);
        resL[1] = p(1);
        resL[2] = p(2);
        resL[3] = p(3);
        resL[4] = p(4);
}EFL
print time.clock() - start_time, "seconds“
print resL
```

```
import sys
import time
def p(workerId):
        x1=range(1000)
        for i in range(10000):
                for j in range(1000):
                        num1 = j + workerId
                        x1[j] += num1
        return (workerId, sum(x1))
start_time = time.clock()
resL = range(5)
EFL{
        for (i = 0; i < 5; i = i + 1) {
                resL[i] = p(i);
        }
}EFL
print time.clock() - start_time, "seconds“
print resL
```

```
import sys
import time
def p(workerId):
        x1=range(1000)
        for i in range(10000):
                for j in range(1000):
                        num1 = j + workerId
                        x1[j] += num1
        return (workerId, sum(x1))
start_time = time.clock()
resL = range(5)
EFL{
        resLout = maploop(resL, p);
}EFL
print time.clock() - start_time, "seconds“
print resLout
```

```
import sys
import time
def p(workerId):
        x1=range(1000)
        for i in range(10000):
                for j in range(1000):
                        num1 = j + workerId
                        x1[j] += num1
        return (workerId, sum(x1))
start_time = time.clock()
resL = range(5)
resLout = range(5)
EFL{
loop (i = 0):
{
        if (i < len(resL))
                loop(i + 1);
        resLout[i] = p(resL[i]);
}
}EFL
print time.clock() - start_time, "seconds“
print resLout
```

Conclusions

The programming constructs of EFL (a parallel programming tool for creating safe and efficient parallelism) were presented here. EFL programming examples were shown.

Further materials (including installation kits for Linux) can be found at the Flexcomp (Flexible Computation) Research Lab site: http://flexcomp.jct.ac.il

Flexcomp Lab members may be reached at flexcomp@jct.ac.il