EFL:
An Embedded Language for Safe and Efficient Parallel Execution

M. Goldstein [a,b], D. Dayan [a,b], M. Popovic [c], D. Berlowitz [a], O. Berlowitz [a], M. Rabin [a], R. B. Yehezkael [a]

(a) Department of Computer Science, Jerusalem College of Technology, Jerusalem 91160, Israel.
(b) The Programming Instruction Unit, The Hebrew University of Jerusalem, Jerusalem 91904, Israel.
(c) Faculty of Technical Sciences, University of Novi Sad, Fruskogorska 11, 21000 Novi Sad, Serbia.

Motivation

Due to the *heterogeneity of parallel programming platforms* today (MPI, openMP, Python's Threads and Multiprocessing modules), there is a need for a *common approach* which:

1. will free programmers from platforms' technical intricacies, and
2. will allow *flexible computation*, in which sequential and parallel executions produce identical *deterministic* results.

Objectives

Our objective has been to develop EFL, a language which implements this common approach. With EFL we aim to implement safe and efficient parallel execution, in software, hardware, or both.

Method

EFL has been designed as a deterministic parallel programming language in order to guarantee safety of parallel executions.

EFL blocks of code are embedded into a sequential host language program.

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

```
#
… sequential code written in the host language …
#
EFL{
If (a > b) {
        x = f(a);
} else {                    # parallel parts of the code written in
        y = f(a);           # the embedded language
}
x = g(b);
}EFL
#
… sequential code written in the host language …
#
```

EFL blocks:

1. may be embedded into code written in any sequential host language.
2. will be translated into parallel host language code

An EFL pre-compiler has been implemented to do so.

Our first implementation is for Python.

EFL Programming Model

The *EFL Programming Model* imposes three kinds of restrictions 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 building blocks

Basic Constructs	Parallel Constructs
1. *Assignment*-block	1. *LogLoop*-block
2. *If*-block	2. *Loop*-block
3. *Pif*-block	3. *MapLoop*-block
4. *For*-block	

EFL Execution Model

Flexible Computation is a basic concept of the *EFL Execution Model*.

A key aspect of Flexible Computation is *well-defined flexible execution* : parallel and/or sequential program execution orders of a program written according to the EFL Programming Model, will yield identical values.

EFL Implementation

We made changes to the Python Pools in order to allow hierarchical levels of parallel execution.

Translating EFL blocks to parallel Python

```
from examplefuncs import  *
def main():
  ….
  EFL{
    for(i =0; i<p ; i= i+1)
    {
     arrout[i] = multPar(i,q,n,matrix,vec);
    }
    tmpArr[g()] = f(1, 2);
    tmpArr[h()] = f(4, 2);

    b = f(3, b = 4);
    c = f(a = 5, b = 6);
  }EFL
  ….
if __name__ == '__main__':
 main()
```

EFL Pre-Compiler

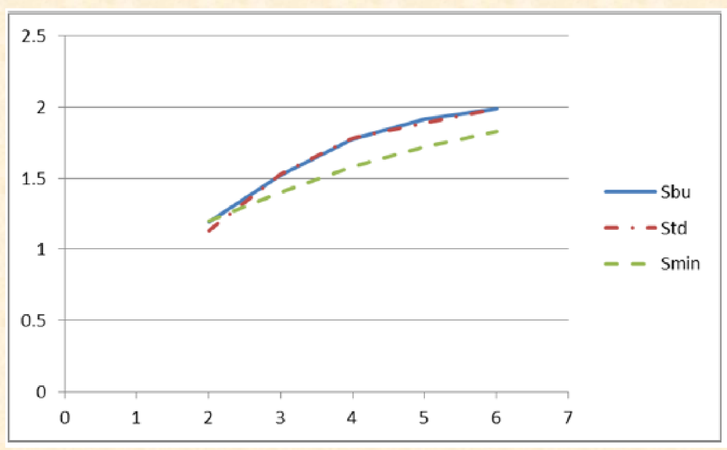
```
from examplefuncs import *
import poolNonDaemon, multiprocessing, math, inspect
Import subprocess, sys
#
def EFL_LOG_LOOP(EFL_LL_LOCAL_QUEUE):
  ….
def EFL_For1(i,vec,n,matrix,q,queue):
  EFL_pool = poolNonDaemon.Pool()
  EFL_ANON_1 = \
        EFL_pool.apply_async(multPar, args = [i, q, n, matrix, vec])
  returnedValue = EFL_ANON_1.get()
  queue.put((i,returnedValue))
  EFL_pool.close()
  EFL_pool.join()
  return returnedValue
def main():
  ….
  # -- Starting EFL Block --
  EFL_pool = poolNonDaemon.Pool()
  manager = multiprocessing.Manager()
  queue = manager.Queue()
  EFL_ANON_3 = EFL_pool.apply_async(g)
  EFL_ANON_4 = EFL_pool.apply_async(f, args = [1, 2])
  EFL_ANON_5 = EFL_pool.apply_async(h)
  EFL_ANON_6 = EFL_pool.apply_async(f, args = [4, 2])
  EFL_ANON_7 = EFL_pool.apply_async(f, args = [3], kwds = dict(b = 4))

  EFL_ANON_8 = EFL_pool.apply_async(f, kwds = dict(a = 5, b = 6))
  i = 0
  while i<p:
     EFL_ANON_2 = \
          EFL_pool.apply_async(EFL_For1,args=[i,vec,n,matrix,q,queue])
  i = i+1
  tmpArr[EFL_ANON_3.get()] = EFL_ANON_4.get()
  tmpArr[EFL_ANON_5.get()] = EFL_ANON_6.get()
  b = EFL_ANON_7.get()
  c = EFL_ANON_8.get()
  EFL_pool.close()
  EFL_pool.join()
  while not queue.empty():
      i,returnedValue = queue.get()
      arrout[i]=returnedValue
  # -- Finishing EFL Block –
if __name__ == '__main__':
 main()
```

EFL validation

The work-span method was used to analyze EFL task trees which enabled us to estimate parallelism and speedup.

Experiments were done to measure actual execution times and calculate real speedups on a specific platform.



Functions of the number of hierarchical levels of parallelism:

S_{bu} - measured *bottom-up* speedup

S_{td}, - measured *top-down* speedup

S_{min} - theoretical *minimal speedup*

Horizontal axis - task tree size

Vertical axis - speedup

Conclusions and Further Work

Conclusions:

An EFL pre-compiler (a parallel programming tool for creating safe and efficient parallelism) has been successfully implemented. Speedup increases were demonstrated by a case study comparing estimated and measured speedup for a family of task trees. Results show that speedup increases almost linearly with the size of a task tree.

Several tracks of further development are in progress:

(a) The EFL pre-compiler is being developed for other host programming languages (C++, Java, C#, Fortran, etc.) and for other platforms (MPI4PY, SCOOP, etc.)
(b) Serial Execution Mode, for debugging purposes.
(c) EFL curriculum, to teach how to implement serial and parallel algorithms using EFL.
(d) IDEs for EFL programming.
(e) Rewriting DEEPSAM with EFL and STM. DEEPSAM is a parallel evolutionary algorithm for protein structure prediction, written in Python upon the TINKER molecular modeling package.