Flexible Algorithms and their Implementation:

An Embedded Flexible Language (EFL)

Max Rabin, D. Berlowitz, O. Berlowitz, M. Goldstein, D. Dayan, R. B. Yehezkael

August 2015 – אב תשע"ה

All authors are with the Flexible Computation Research Laboratory, Jerusalem College of Technology,
Hawaad Haleumi 21, POB 16031, Jerusalem 91160, Israel.
Here are their contact details:
Max Rabin (e-mail: rabin.max@gmail.com)
D. Berlowitz (e-mail: devora32@gmail.com).
O. Berlowitz (e-mail: berlo.berlo@gmail.com)
M. Goldstein (e-mail: goldmosh@jct.ac.il).
D. Dayan (e-mail david674@bezeqint.net).
R. B. Yehezkael (phone: +972 2 6751294; fax +972 2 6432145; e-mail: rafi@jct.ac.il).

***Abstract*—Parallel programming is a very difficult task and there is a necessity to simplify it. Here we describe EFL (Embedded Flexible Language) that gives an answer to this need. EFL is an implementation of the flexible algorithmic language proposed in the companion paper [1]. EFL simplifies parallel programming by embedding blocks capable of flexible computation inside code written in an imperative host language (our first implementation is for Python). A pre-compiler was created to convert EFL embedded blocks of code into optimized native parallel code in the host language. This will allow programmers to take advantage of parallelism without having to learn the intricacies and complications that come along with it. Run-time benchmarking will be provided in order to show the effectiveness of the EFL-approach to parallel programming.**

***Index Terms*— Embedded language, Flexible execution, Scope rules, Composite assignments.**

1 Introduction

In 1965, Gordon E. Moore [2] formulated a “law”, now known as *Moore’s Law,* that the number of transistors that can be placed on a circuit will double every two years. However, at some point, Moore’s law must fail [3] because of physical limitations. In order to maintain computing advances, hardware makers have begun to look elsewhere for speed increases and it is now commonplace to increase the *number* of circuits in a computer. Servers, home computers, and even smartphones now come with multiple CPU cores [4,5]. Parallel computing has been around for about as long as operating systems. There has always been a need for computers to execute multiple programs at the same time [6]. However, due to practical constraints (namely having only one CPU), parallel computing has not been so popular [6]. Threads have been used to make programs with high Input/Output activity more efficient, delegating the task of ‘waiting’ for files or network events to a separate thread so that the main thread (or User Interface thread) does not ‘hang’. However, this does not help the programmer in the case of CPU-intensive operations.

When multiple CPU-intensive operations had to be performed, then parallelizing the code would not yield any performance benefits because of the fact that the code was ultimately running on one CPU and therefore no real parallelization was occurring. Therefore, little advancement has been made in the field of parallel computing.

In recent years, multi-core CPUs have become abundant. Therefore, the need has arisen in recent years for programmers to parallelize CPU-intensive code. This is not a simple task. Whereas serial algorithms and code can be compiled onto any platform, many different types of parallel algorithms exist for many different platforms. There are so many different types of parallel algorithms, some assuming specific CPU configuration, others assuming shared physical memory. These algorithms are not applicable to today's computers. The challenge for today's programmers is even greater. Marowka [7] claims that as opposed to other technologies that are taught in schools, parallel computing is not taught properly. He says that instead of preparing programmers to think in ways that parallel computing requires, we teach serial languages like Java, and are producing a generation of serial programmers who are not equipped to deal with the challenges of parallel programming. Similar opinions are expressed by Brown et al. [8] and by Graham [9]. There are many reasons for this. First of all, parallel programming is much more complex. It requires a completely different way of thinking, and has a whole set of issues that must be taken into account. Debugging parallel programs can be very difficult, and special techniques must be employed. There are some problems, like deadlocks and race conditions, that don't even appear every time the code is run. A program can work perfectly during development but during production, a problem related to parallel execution can occur with no easy way to diagnose the problem [10].

According to Amdahl's Law [11], the extent to which we can speed up any complex computation is limited by how much of the computation must be executed sequentially. The speedup S is defined as the ratio between the sequential (single-processor) time and the parallel time. Amdahl's Law characterizes the maximum speedup *S* that can be achieved by *n* processors collaborating on an application, where *p* is the fraction of the computation that can be executed in parallel. With *n* concurrent processors, the parallel part takes time p/n, and the sequential part takes time $1-p$. Overall, the parallelized computation takes time $1-p+p/n$. Amdahl's Law says the speedup, is $S=1/((1-p)+p/n)$, meaning that the speedup *S* is limited, no matter how many cores are added. In the Embedded Flexible Language (EFL) which we have developed, the parallel parts of a program are written in EFL blocks, in which well defined flexible execution order is implemented, and the sequential parts of the program are written in the host language, outside the EFL blocks. EFL is a simple language, embedded into a host language, that translates into the host language using an EFL Preprocessor.

The idea behind EFL is to create an embedded language that looks and feels like a traditional C-based sequential language, but in fact executes in parallel the code inside EFL-blocks. This enables the average programmer to create parallelized code to optimize

the use of multiple CPUs, in a simple and safe way, that doesn't require the programmer to learn about the complications and intricacies of parallel programming. For example, an object shared between threads, such as a Thread Safe Stack might have atomic `push()` and `pop()` operations. The Thread Safe Stack could be written in a traditional language like so:

```
class ThreadSafeStack {
    Stack stack = new Stack();
    Lock lock = new Lock();
    void push(Object value) {
        lock.acquire(); //Program waits here until lock is acquired
        stack.push(value);
        lock.release();
    }

    Object pop() {
        lock.acquire(); //Program waits here until lock is acquired
        Object ret = stack.pop();
        lock.release();
        return ret;
    }
}
```

In this example, parallel threads can push data to and receive data from the Stack without worrying about synchronization issues. Once the lock is acquired by one thread, any other thread that tries to interact with the Stack will wait. We have essentially turned parallel communication into a sequential process. This can also cause problems of deadlock and starvation [12]. This type of situation can be very common in parallel programming and it is exactly what we want to avoid when programming a parallel program. EFL creates an environment for true parallel programming, without the complications thereof. An EFL Preprocessor can be implemented for any host language, allowing programmers to learn EFL only once, using it for the parallel needs of their applications, in different languages.

EFL was created to enable the implementation of Flexible Algorithms and their well defined order-independent execution, as described in the paper [1].

1.1 Layout of the paper

The remainder of this paper consists of the following sections:

2. THE EFL LANGUAGE
3. EFL GRAMMAR
4. THE HOST LANGUAGE PYTHON
5. COMPILER IMPLEMENTATION
6. FUTURE WORK

2 THE EFL LANGUAGE

When deciding how to build the EFL language, the following considerations were taken into account: (a) easy to learn, (b) familiarity to programmers, and (c) easy to parse. The purpose of EFL is to provide programmers with an intuitive tool that does not require extensive training so they could utilize their familiarity with other languages when learning EFL. Through familiar constructs such as `if` conditions, `for` loops, and function calls, a programmer with experience in traditional languages should have no problem getting started in EFL. There are some limitations in EFL that will be new to programmers. Those limitations are imposed on the language due to the requirements of Flexible Algorithms' principles and their parallel implementation. However, it is apparent the resemblance between EFL's and C's syntax. For example, the following EFL code calculates the factorial of a collection of numbers:

```
for (i = 0; i < len(inArray); i = i + 1)
{
    outArray[i] = factorial(inArray[i]);
}
```

This code would compile in C, C++, Java, C# and Javascript and it would do the same thing in each of those languages. No doubt, a programmer familiar with any of these C-syntax-based languages can tell that the semantics of this for-loop is to loop over the `inArray` by incrementing `i` by one at the end of each iteration instance. In each iteration instance of the loop, the `factorial` function is called with the current item of `inArray`. The value returned from `factorial` is stored in the corresponding index of the `outArray`. What EFL does, though, is fundamentally different than other C-syntax-based languages. The EFL semantics of the same `for` loop, though, is as follows: instead of looping through each iteration instance of the loop, one at a time, proceeding to each subsequent iteration instance only after finishing the body of the current iteration instance, EFL executes each iteration instance of the loop *in parallel*. Each of the calls to the `factorial` function is executed in parallel, utilizing the multiple cores of the computer executing the program. In addition to the iterative constructs familiar to the programmer, some additions were made to allow implementation of advanced parallel algorithms.

EFL can only be placed within functions that are declared in the host language. This was decided for the portability of EFL to different languages. Different languages define functions in different ways, a difference which is particularly felt is the difference of typed versus untyped languages. A typed language will define a function with a return value and will require that all parameters be defined with a type. On the other hand, untyped languages don't require return types for function declarations or

parameters. By not including function definitions in EFL, the language can be kept host agnostic, and thus EFL code can be ported from one host language to another without modification. EFL code exists within the context of the host language, with full accessibility to the host function's variables, context, and scope.

2.1 Limitations

The theoretical foundations of the limitations enforced by EFL in order to ensure deterministic parallelization are described in detail in the companion paper [1]. Essentially, those limitations are of three kinds: (a) only "pure" function calls, (b) *In* and *Out* variables (and not *InOut* variables!), and (c) once-only assignments.

"Pure" functions – a "pure" function has no side-effects. That means no global variables, no Input/Output, etc. "Pure" functions only perform calculations and, thus, always produce and return a value. Without a return value, the function does not provide any utility to the program [13]. Therefore, functions called from an EFL block cannot be a `void` function because a `void` function can change global items relative to the function. This limitation was imposed in order to enable well defined parallelism. If for example, two parallel function calls were to append "A" and "B" respectively to a file, the resulting file might contain "AB" or "BA", a situation of non-deterministic output. This is because there is no guaranteed order of execution with parallel processes. By allowing only "pure" functions, there is no chance for different threads of execution to cause conflict.

In and *Out* variables – let's define a variable whose value is read within an EFL block as *In*, meaning that it is an input to the block; similarly, a variable to which a value is assigned from within an EFL block is designated as *Out*, meaning that the value is an output of the EFL block. An *In* variable cannot be written to and an *Out* variable cannot be read. This limitation prevents problems of race conditions [12] and nondeterministic execution. If the ability to read and write to all variables was free and unbounded, a situation could arise where at one point in the code the programmer assigns a value to a variable,

```
x = f(a);
```

and then reads that variable in another subsequent line,

```
y = f(x);
```

This would be legal in sequential languages, but in EFL this would be problematic because the second line is executed in parallel with the first and therefore the order of execution is unknown, so the value of `x` when evaluated in the second line may have different values, depending on the order of execution. In order for the execution of EFL code to be deterministic, this could not be allowed. Therefore if a variable is assigned to anywhere in an EFL block it is designated as an *Out* variable and its value cannot be read inside the EFL block. Likewise if a variable is read inside an EFL-block, it is designated *In* and cannot be

assigned to anywhere in the EFL-block. The EFL Pre-compiler identifies the *In* and *Out* variables and notifies the programmer of any violations of this rule.

Once-Only Assignments – it is a rule that comes out of the *In* and *Out* variable rule. Once a variable is designated as an *Out* variable, it does not make sense to assign to it multiple times in the same EFL block because each assignment is applied in parallel, and the order of the assignments is unknown. Once the order of assignments is unknown, the final value of the variable is also unknown. In an imperative language, the following code:

```
x = 5;
x = 6;
```

would yield a value of 6 for the variable `x`. In EFL this is not necessarily true. The variable `x` could attain the value 5. Therefore the results of this program would not be deterministic, so multiple assignments to the same variable are prohibited. Violations of once-only assignment cannot be checked at compile time. If it were possible to know at compile time if a variable has been assigned a value, it would be possible to solve the Halting Problem [14]. This has been proven to be impossible. To demonstrate this point, consider the following code:

```
EFL{
if (a > b) {
    x = f(a);
} else {
    y = f(a);
}
x = g(b);
}EFL
```

Has a violation of the once-only assignment rule occurred? At first glance it would appear that `x` is assigned to twice, but in fact the first assignment only occurs in the case where `a > b`. If `a` is not greater than `b`, no violation of the once-only assignment rule has occurred. Therefore, a checking mechanism was implemented to produce code in the host language that checks for violations of once-only assignments. Violations of the once-only assignment rule are reported at runtime. Although, a violation of once-only assignment is considered invalid according to the definition of EFL, the checking mechanism can be excluded from the compiled code with a special flag passed to the compiler. This allows for more experienced programmers to produce efficient code and use the once-only assignment checker as a development tool.

2.2 Basic Constructs

One of the driving thrusts of EFL was to create a language with semantics that are similar to sequential languages. It was important to have as small of a learning curve as possible, to encourage adoption by programmers and teachers. The basic constructs of EFL are: (a) Assignment Block, (b) If Block, (c) PIf Block, and (d) For Block.

Assignment Block – Similar to C, it begins with the variable being assigned to followed by the assignment operator (the equals sign) followed by the value being assigned to the variable. For example:

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

Whereas in C these two statements would execute sequentially, in EFL they are executed in parallel. The benefit of this is particularly felt when used with function calls that are CPU-intensive. For example:

```
EFL{
myVal1 = cpuIntensiveOperation(someParameter);
myVal2 = cpuIntensiveOperation(someOtherParameter);
}EFL
```

The two calls to the `cpuIntensiveOperation` function are executed in parallel. This basic example of EFL exemplifies the core power of EFL, executing long-running functions in parallel. Values can be assigned to variables or into a specific index of an array. If the index of the array is represented as a statement with a function call, for example `arr[f(x)]`, then `f(x)` is evaluated in parallel. Likewise, if a function is called with values which are a result of function calls, then those are calculated in parallel but not the 'outer' function call. Note that the evaluation of such expressions in EFL is similar to the substitution model of expression evaluation in functional programming languages like Scheme [15]. In the following example, all of the calls to `f` are calculated in parallel:

```
arr[f(a)] = g(f(x) * f(y), f(z));
```

Only after all the calls to `f` return their result values, `g` is called.

If Block – Like in C, it is a construct allowing for conditional execution of code. It allows for alternative conditions using the keyword `elseif`, and a default condition `else` that is executed when none of the conditions result in `true`. A typical `if` block can look like the following:

```
EFL{
if ( conditionOne(someValue) ) {
    // some code here
} elseif ( conditionTwo(someValue) ) {
    //other code here
} elseif ( conditionThree(someValue) ) {
    //other code here
} elseif ( conditionFour(someValue) ) {
    //other code here
} else {
    //last code here
}
}EFL
```

In terms of functional programming, the `if` statement in EFL is similar to a Special Form - although function calls in EFL are usually executed in parallel, a function call in the condition of an `if` statement is not executed in parallel. That way, we can execute the body of the `true` block in parallel to the rest of the EFL block. If the conditions were analyzed in parallel to the rest of the EFL-block (like they are in `pif`, see below), then the decision of what block to execute would have to be delayed until the results of all the conditions are determined. This would create a barrier in the parallel execution of the EFL-block. Therefore, we decided that the conditions of an `if` statement are not analyzed in parallel. If the programmer would like to utilize parallelization to analyze `if` conditions, then he has two options: (1) to use a `pif` (see below), or (2) to write an EFL-block in which the necessary conditions are calculated in parallel and the results are saved to a variable, and then, to use the results in a subsequent EFL-block. This could look like the following:

```
EFL{
a = f(x);
b = g(x);
}EFL
EFL{
if (a) {
    //some code
} elseif (b) {
    //some more code
} else {
    //other code
}
}EFL
```

That way, the programmer has more fine grained control over the conditions. It is important to note that this is not a violation of the *In* and *Out* variables rule. When `a` and `b` are assigned to in the first EFL-block, they are established as *Out* variables. However, that only applies to the scope of the first EFL-block. In the next EFL-block, their *In* / *Out* scope is reset and when they

are encountered there, their values are read, thus defining them as *In* variables. This is perfectly valid and in fact encouraged in EFL.

PIf Block – It is similar to an *if block* in that it provides a mechanism for conditionally executing code. Using the keyword `pif`, this construct allows for all of the conditions of the *if* to be evaluated in parallel. The *pif* computes the conditions of a `pif elseif else` in parallel so that by the time all of the conditions' values are calculated, the block controlled by the first `true` condition could be jumped to and executed. There are certain issues that must be taken into account with this approach. Take for example the following code:

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

In a serial language, there would be no chance of a 'divide by zero' exception when executing this code, because the only way that the second condition, `((b / a) == c)` would be evaluated is if `a` does *not* equal zero. However, in a parallel execution, when all conditions are evaluated in parallel, then regardless of the result of the first condition, the second condition would also be evaluated. If in fact `a` *does* contain the value zero, then this would result in a run-time error.

One downside of the `pif` is that it adds a blocking phase inside of the execution of EFL. By design, every statement in an EFL block is executed in parallel. However, with a pif, we must first wait until all of the conditions are evaluated, and only then can we execute the block decided by the conditions. That means that not every statement in the EFL block is evaluated in parallel, but instead in a series of parallel computations. In order to avoid these blocking conditions, a programmer can choose to only use `if` statements, and avoid `pif`s altogether. Although the `pif` bends one of the design rules of EFL, we felt that this option should be available to programmers to use, and will allow for sufficient parallelism in EFL programs.

For Block – It is a loop construct allowing for a block of code to be executed multiple times. Due to the once-only assignment restriction, *For blocks* are only useful when used to loop through arrays. Each iteration instance must save any calculated values to a different slot in an array. That way, there will be a well defined result at the end. For example:

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

By assigning to different indices of the array in parallel, we can ensure deterministic results. Assigning to a different index of the array in each iteration instance avoids violations of the *once- only assignment* rule. As a side note, the variable `i` violates both the *In* and *Out* variable rule and the *once-only assignment* rule. It is assigned to in the initialization of the loop, defining it as an *In* variable. After the body of the loop executes, the value of `i` is incremented, violating the *once-only assignment* rule. Its value is subsequently read in the condition of the loop to decide if the loop should continue to the next iteration instance or not. Reading the value of `i` defines it as an *In* variable. Having already been defined as an *In* variable, `i` violates the *In/Out* designation rule. Because those rules are only a safeguard against non-deterministic results of a parallel execution, and the variable `i` is never actually assigned to or read in parallel in the processed code in the host language, this exception to the rules was allowed. It was decided that for the benefit of the programmer, the `for` block must remain as similar to the C `for` block as possible.

2.3 Parallel Constructs

Although EFL was designed to mirror a sequential language as closely as possible, there were some constructs added to the language that make use of the parallel nature of the language: (a) LogLoop Block, (b) Loop Block, and (c) MapLoop Block.

LogLoop Block – It was created in order to allow the developer to perform an efficient parallel scan. This is a parallel algorithm that executes an operation on all values of an array. To do this, we will use an algorithmic pattern that arises often in parallel computing: balanced trees. The idea is to build a balanced tree on the input data and sweep it to the root. A `k`-ary tree with `n` leaves has $log_k(n)$ levels and therefore the running time of the algorithm is $O(log_k(n))$ where `k` is the number of parameters of the function, and `n` is the length of the array. One example function is addition. The algorithm works by summing every two elements in parallel, then summing every two resulting values and so on. This will produce the sum of all values of the array. The *LogLoop Block* requires the programmer to specify the function to apply and the collection on which to apply the function. Note that the *LogLoop Block* is similar but more general than the high-order function *Reduce*, common in functional programming languages. A call to *LogLoop* looks like this:

```
EFL{
result = logloop(myArray, myFunction);
}EFL
```

The number of input elements for the function is dynamically detected at run-time and the algorithm acts accordingly. If the function accepts three parameters, then the running time of the algorithm is $O(log_3(n))$. The following pseudo code illustrates the execution in the background:

```
i = 0
for h = 1 to log2(n)
{
```

```
   i = h
   while (i   <    = (n / 2^h))
   {
     z = Array[i] op Array[i+1]
     Array[i+1] = z
     i += 2
   }
}
```

A graphical view of the execution of the algorithm performing addition on the numbers 1 through 8 looks like this:

```
round 0: 1  2  3  4  5  6  7  8
          \ |   \ |   \ |   \ |
round 1: 1  3  3  7  5 11  7 15
             \    |      \    |
               \  |        \  |
                 \|          \|
round 2: 1  3  3 10  5 11  7 26
                   \          |
                     \        |
                       \      |
                         \    |
                           \  |
                             \|
round 2: 1  3  3 10  5 11  7 36
```

At the end of round 3 ($log_2(8) = 3$), we have the result 36.

Loop Block – It is an abstraction of loops that decouples the repetitive nature of a loop from the order of execution. Whereas traditional loops like `while` and `for` work sequentially, EFL executes loops in parallel. These traditional loops are syntactic sugar around a sequential execution of code with a `goto` command at the end of the statements of the loop. A `while` loop in C is equivalent to

```
BEGIN_LOOP:
    if (not(condition)) {
        goto END_LOOP;
    }
    //statements of the loop
    goto BEGIN_LOOP;
END_LOOP:
```

In EFL, in order to emphasize the decoupled nature between the order of execution of the loop and the body of the loop, the *Loop Block* was created. Here is an example:

```
:loop (i = 0)
{
    if (i < len(arrIn)) {
        loop(i + 1);
    }
    arrOut[i] = cpuIntensiveFunc(arrIn);
}
```

The *Loop Block* is declared with an identifier and an initial value for the iteration variable. That identifier (`loop` in the previous example) is defined by the programmer, and is not a language reserved keyword. Within the body, a recursive call is made to the loop using the identifier, and passing a new value for the new iteration variable. This provides the ability to advance the iteration using varying increments of the iteration variable. The *Loop Block* is "syntactic sugar" for a function declaration, and is looped through using a recursion-like syntax. This promotes a lambda expression style of programming, in which functions are declared "on the fly" and executed on the spot. This style of programming is common in functional programming [16] and especially suited for parallel programming [17] where all iterations of the loop is executed in parallel. Traditional loops, which are just fancy goto statements to jump around code, promote the idea that the body of the loop is a series of instructions that get executed one after another. Alternatively, the loop block views the body of the loop as a whole unit, which is executed all at once and in parallel to other iteration instances.

MapLoop Block – This construct is based on the "map" function of functional programming languages [15, 18]. It applies a function to every element of a collection. In functional programming this is a very common pattern due to the immutable nature of variables. In a functional language, variable value or state never changes. Instead, calculations are made and results are returned and saved into new variables and then the work continues using those new variables. If a programmer wants to square every element of an array in a functional language, then every element is squared and the results are stored in a new array, not in the original one. This makes the map function especially useful, because it does not change the values upon which the map is called, rather it returns the result of the function on those numbers. This is trivial to parallelize because the execution of the function with a given element in the collection is independent of the execution of the function with a different element in that collection, and order of execution is irrelevant. If there were a function called `square` which received a number as a parameter and returned the mathematical square of that number, it could be applied to an array called `myArray` with the following statement:

```
EFL{
    squares = maploop(myArray, square);
}EFL
```

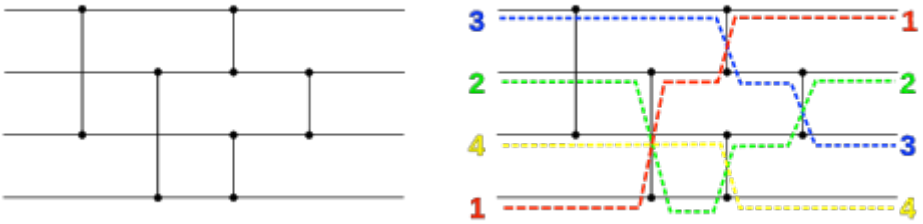
Although in general EFL was designed to mimic a sequential imperative language like C or C++, the *MapLoop* was included because it helps encourage thinking about loops in a non-sequential way.

2.4 Parallel Algorithms

EFL can be used to implement parallel algorithms that use comparator networks in a very intuitive way. Sorting Networks are abstract mathematical models of networks of wires and comparator modules that are used to sort a sequence of numbers. In a sorting network [19], each comparator connects two wires and sorts the values by putting the lower value on one wire and the higher value on another wire. A sorting network is built in a static way so that the same operations are performed regardless of the result of the previous comparisons.

A simple sorting network consisting of four wires and five connectors

Each comparator of a particular step executes concurrently. The time of the algorithm is evaluated as the number of steps in the network ("depth"), and the number of comparators is the "cost" of the network. Here is a demonstration of the simple sorting network presented above.



A simple sorting network in action

To implement this sorting network in EFL we would convert each step into an EFL block. The contents of each EFL block are executed in parallel, but each EFL block waits for the previous one to finish, because the host language is sequential.

Any sorting network can be easily translated into EFL as follows.

```
def compare_swap(x, y):
```

```
    if ( x <= y ):
        return x, y
    return y, x
a = toSort[0]
b = toSort[1]
c = toSort[2]
d = toSort[3]
EFL{
a1, c1 = compare_swap(a, c);
b1, d1 = compare_swap(b, d);
}EFL
EFL{
a2, b2 = compare_swap(a1, b1);
c2, d2 = compare_swap(c1, d1);
}EFL
EFL{
a3 = a2;
b3, c3 = compare_swap(b2, c2);
d3 = d2;
}EFL
sorted = [a3, b3, c3, d3]
```

Based on the above, it is interesting to note that our notion of EFL-block is an ideal platform for writing programs that resemble the *Bulk Synchronous Parallel Model* (BSP) introduced by Leslie Valiant in 1990 [20]. Each block of EFL executes its contents in parallel, that play the role of BSP *components*. After the EFL block, all of the values that were computed inside the EFL block, are returned as OUT variables, and are accessible in the main program which acts as the BSP *router*. Also, because the execution of an EFL block will wait until all processes have finished executing, the closing of the EFL block represents the BSP *synchronization* facility. Thus, all what is executed inside an EFL block may be viewed as a *superstep* in BSP terminology.

3 EFL GRAMMAR

The Grammar of the EFL language, represented in EBNF notation, is included here for reference.

```
S := (Start Statements End)* <EOF>

Start := 'EFL{'

End := 'EFL}'

Statements : = Statement*

Statement := ( <ExtCallBlock>
             | <LogLoopBlock>
             | <MapLoopBlock>
             | <AssignmentBlock>
             | <IfBlock>
             | <PifBlock>
             | <LoopBlock>
             | <ForBlock>
```

```
              | <SubroutineCall>
              )

  ExtCallBlock := <VarName> <EQL> <EXTCALL> <LPAREN>
                  <STRCONST> ',' <STRCONST>

EQL := '='
  LogLoopBlock := <VarName> <EQL> <LOGLOOP>
                  <LPAREN>
                     <VarName> <COMMA> <SubroutineName>
                  <RPAREN> <SEMI>

  MapLoopBlock := <VarName> <EQL> <MAPLOOP>
                  <LPAREN>
                      <VarName> <COMMA> <SubroutineName>
                  <RPAREN> <SEMI>

  AssignmentBlock := <VarName> ( <LREC> <Expression> <RREC> )?
                     ( <COMMA> <VarName>
                         ( <LREC> <Expression> <RREC> )?
                     )*
                     <EQL> <Expression> <SEMI>

  LoopBlock := <COLON> <IDENTIFIER>
               <LPAREN> <VarName> <EQL> <Expression> <RPAREN>
               <LCURL> <Statements> <RCURL>

  IfBlock := <IF> <LPAREN> <Expression> <RPAREN>
             ( ( <LCURL> <Statements> <ECURL> )
             | <Statement>
             )
             ( <ElseIfBlock> )*
             ( <ElseBlock> )?

  ElseIfBlock := <ELSEIF> <LPAREN> <Expression> <RPAREN>
                 ( ( <LCURL> <Statements> <ECURL> )
                 | <Statement>
                 )

  ElseBlock := <ELSE> ( (<LCURL> <Statements> <ECURL> )
                      | <Statement>
                      )

  PifBlock := <PIF> <LPAREN> <Expression> <RPAREN>
              ( ( <LCURL> <Statements> <ECURL> )
              | <Statement>
              )
              ( <ElseIfBlock> )*
              ( <ElseBlock> )?

  ForBlock := <FOR>
              <LPAREN>
                  <ForInit> <SEMI>
                  <Expression> <SEMI>
                  <ForUpdate>
              <RPAREN>
              ( ( <LCURL> <Statements> <ECURL> )
              | <Statement>
              )

  ForInit := ( <VarName> <EQL> <Expression> )*

  ForUpdate := ( <VarName> <EQL> <Expression> )*

  SubroutineCall := <SubroutineName> <LPAREN>
                    ( <ExpressionList> )?
                    <RPAREN>

  Expression := <Term> ( <OP> <TERM> )*

  TERM :=   <SubroutineCall>
          | <NUMCONST>
          | <KEYWORD_CONSTANT>
          | <LREC> ( <Expression> )? <RREC>
```

```
        | <VarName> ( <LREC> ( <Expression> )? <RREC> )?
        | <LPAREN> <Expression> <RPAREN>
        | <PRE_UNARY_OP> <Term>

ExpressionList:= <SpecialExpression> ( <COMMA> <SpecialExpression> )*

SpecialExpression := ( <VarName> <EQL> <Expression> )
                   | <Expression>

VarName := <IDENTIFIER>

SubroutineName := <IDENTIFIER>

IDENTIFIER := ( ['a'-'z']
              | ['A'-'Z']
              | '_'
              )
              ( ['a'-'z']
              | ['0'-'9']
              | ['A'-'Z']
              | '_'
              )*

EXTCALL := "extcall"
```

4 The Host Language Python

4.1 Why Python

EFL was designed as a language to introduce to parallel programming and make it easier. Our implementation of EFL uses Python 3.x [21] as its host language. There were several factors taken into consideration when deciding which host language to use: (a) We wanted a host language that is easy to learn, and Python was a good candidate in this respect. This would place EFL as an ideal language to be taught to new programmers as explained in [22]. (b) Another factor considered was cross platform compatibility. It was important to choose a language that anybody can use on any platform so that nobody is precluded from using EFL because of the operating system they run. (c) Another factor considered was the parallel capabilities of the candidate host language. The functional nature of Python, along with the robust and easy to use Threading and Multiprocessing libraries, provides an easy to use platform for creating parallel code [23, 24].

4.2 Implementing Parallelism

There are two modules built into the standard Python core that allow parallelism - threading and multiprocessing. The threading module creates threads at the operating system level using the POSIX thread (also known as “pthread”) implementation. Unfortunately, CPython (the implementation of the Python language specification most commonly used) has a lock mechanism called the Global Interpreter Lock (GIL) which allows only one thread to execute at once. That means that even on a system with multiple CPUs only one thread can be executed at once. This mitigates any benefit that can be derived by parallel computing. The threading module is useful for I/O bound tasks that have multiple concurrent threads waiting for I/O events, but not for the purposes of parallel computations. Therefore, we use the multiprocessing module to implement parallel computations. The

multiprocessing module uses sub-processes instead of threads to achieve parallelism. Although only one thread can execute Python code due to the GIL, multiple processes with one thread each, may run concurrently on multiple CPUs [25]. Therefore, the multiprocessing module was used to implement the parallelism of EFL. Within the multiprocessing module there are different methods for creating and working with processes. One way is using the Process class to create a Process object that is set to execute a function, and calling its start() method to begin the execution. In order to pass arbitrary variables to the function using the Process class, we would have had to use Queues to pass parameters to the function. The other option is to create a Pool of processes. A Pool is a collection of a fixed number of processes. The number of processes in a Pool are defined when the Pool is created. The number of sub-processes defaults to the number of CPUs on the computer. Another benefit of the Pools is that parameters can be passed without using a Queue. The Pool class also includes map functionality allowing simple implementation of EFL's maploop. This will apply a function to every element in an array in parallel. One of the biggest deficiencies of Pools is that the sub-processes of the pool are 'daemonic' processes. Daemon processes are created by a parent non-daemonic process and are able to continue to run even if the parent process ends. However, daemonic processes cannot spawn sub-processes. That means that the functions executed by a Pool cannot themselves spawn child processes. Processes created with the Process class can be created either as daemons or not daemons. This difference ends up having a very significant impact on the abilities of EFL. If we use Pools as the mechanism of parallelism in EFL, then EFL blocks cannot contain function calls that have EFL blocks within them. That is because the functions called from within EFL are executed by a Pool and therefore, are run as daemon processes. This is means that they cannot contain another EFL block which would be trying to create a new Pool of processes. This would limit EFL to one level of child-processes and no more. However, if we were to use the Process class, we would be able to run functions called by EFL in a non daemonic process and they would in turn be able to contain EFL blocks that also create child processes. This would allow a depth of EFL within EFL that executes all of the processes and sub-processes in parallel. This would be ideal in order to allow the programmer the freedom to embed EFL in any level of a program with no regard to how any particular function is called - by EFL or by the host language. This freedom of Processes would be the ideal way to implement EFL. However, in reality, there is a limit on the number of sub-processes that an operating system can handle. Although an operating system is responsible for managing all of the processes running on a computer, there are limits on the number of concurrent processes that an operating system can manage and there are performance considerations that must be considered. The performance of the applications worsens considerably when the number of processes exceeds the total number of processes that the system can handle and the number of processors. Furthermore, the larger the number of processes the worse the performance gets. The decreased performance in parallel applications, when the number of processes in the system exceeds the available number of processors, can be attributed to several problems. These problems include the following: (a) the frequent

context switching that goes on when the number of processes greatly exceeds the number of processors; (b) aside from the problem of corrupted caches (discussed below), a context switch involves a number of system-specific operations that do no real work. Too many processes will start to fill up the computer's memory [26]; (c) another source of performance degradation is the issue of starvation. With Pools, this is not an issue because the Pool is created with a fixed number of child processes (and not specifying a number of children makes the Pool default to create a number of child processes equal to the number of processors). Therefore no starvation occurs because the Pool lets each task assigned to it finish before executing the next task assigned to it. In order to decide whether to use Processes or Pools, we did a simple test that executes a task in parallel using Processes, and using Pools, and we compared the running time of the two. We ran an algorithm on an n * n array where an operation (finding whether the number is prime) is executed for each element of the n * n array in parallel. This is the program:

```
import multiprocessing, collections, time

def testPool(bigNum):
    pool = multiprocessing.Pool()
    processes = collections.deque()
    bigArr = [[0] * bigNum] * bigNum
    for i in range(bigNum):
        for j in range(bigNum):
            processes.append(pool.apply_async(f,
                                              args = [i * bigNum + j]))
    for i in range(bigNum):
        for j in range(bigNum):
            processes.popleft().get()

def testProcesses(bigNum):
    processes = collections.deque()
    for i in range(bigNum):
        for j in range(bigNum):
            p = multiprocessing.Process(target = f,
                                        args = (i * bigNum + j,))
            processes.append(p)
            p.start()
    for i in range(bigNum):
        for j in range(bigNum):
            processes.popleft().join()

def main():
    bigNum = 100

    start_time = time.time()
    testPool(bigNum)
    elapsed_time = time.time() - start_time
    print("Using Pool:    ", elapsed_time)
```

```
    start_time = time.time()
    testProcesses(bigNum)
    elapsed_time = time.time() - start_time
    print("Using Process: ", elapsed_time)

if __name__ == '__main__':
    main()
```

The execution times were as follows:

Input Size	**Process (secs.)**	**Pool (secs.)**
2 * 2 (4)	0.269999980927	0.15700006485
10 * 10 (100)	10.75	0.153000116348
15 * 15 (225)	22.4300000668	0.174999952316
100 * 100 (10,000)	Crashed	3.25500011444

Even with a 15 by 15 array, using Process is drastically slower, and on a simple 100 by 100 array, the operating system isn't even capable of handling this many processes. Given the above, although we would have preferred to use Processes because of their flexibility. However, the constraints of the reality forced us to use Pools so that programmers would be able to utilize one level of EFL as they wish, irrespective of the number of processes in the pool. In the end, in order to truly get the flexibility of Processes along with the safety and efficiency of Pools, we implemented our own custom Pool class, based on the Pool class that ships with Python. Our modified Pool creates child processes that are not daemonic processes. That way, child processes of the Pool can themselves create Pools, allowing in principle an unlimited depth of parallelism, which is ideal for the EFL language. It also has the safety that Pools afford, in that it only creates a fixed, limited number of child processes when the Pool is created, and each asynchronous job that is added to the Pool is queued until a child process is able to execute it. This was an achievement that even gives EFL an advantage over a Python programmer who manages the processes without the help of EFL. If, however, the programmer creates too many child levels of processes, the number of processes can still grow exponentially like they did when we tried to use Processes, which can still crash the computer executing the program. However, this exponential growth is limited by the number of Processes per Pool, thus the growth is slowed and most programs should suffice with this implementation.

The Python code generated by the EFL preprocessor consists of two phases. The first phase is the spawning of parallel tasks and the second waits for the tasks to end and collects the results. This structure provides the framework for code with any depth. If an ‘assignment’ block appears within a ‘for’ block then the ‘for’ gets translated into two ‘while’ loops in python, with the two parts of the ‘assignment’ blocks within. The first ‘while’ contains the code that spawns the parallel tasks for the ‘assignment’ block and then the second ‘while’ contains the code that collects the results from the ‘assignment’ tasks. For example, the following code

```
EFL{
for (arrIndex = 0;
     arrIndex < len(inputArray);
     arrIndex = arrIndex + 1) {
    outputArray[arrIndex] =
        cpuIntensiveFunction(inputArray[arrIndex]);
}
}EFL
```

would translate into Python in a manner similar to this simplified pseudo-code

```
//spawn tasks to be executed in parallel
arrIndex = 0
while arrIndex < len(inputArray):
    parallelHandle[arrIndex] =
        spawnParallelTask(cpuIntensiveFunction, inputArray[arrIndex])
    arrIndex = arrIndex + 1
//collect results
arrIndex = 0
while arrIndex < len(inputArray):
    outputArray[arrIndex] = getParallelResult(parallelHandle[arrIndex])
    arrIndex = arrIndex + 1
```

In this way, it was possible to translate EFL into parallelized code.

It is important to note that the nature of the `multiprocessing` library is to delegate a function to a process. It was decided that only certain statements within EFL would actually be parallelized with processes. For example, two assignments

```
EFL{
a = 5 + 5;
b = 6 + 6;
}EFL
```

would not be worth parallelizing. The overhead incurred on each parallel call is too great and in order to optimize EFL, only function calls are parallelized. Therefore, in order to maximize EFL’s effectiveness and efficiency, it is best used for function calls to CPU intensive operations. Simple, less CPU intensive functions should be avoided in EFL and are best called from

Python directly. Only parallelizing function calls also solves many issues of scoping, because the EFL is translated on the spot inside of the host language where it is written. That way, any variables or functions that exist within the lexical scope where the EFL is defined are also present where the EFL is translated. In order to take statements that are not function calls and parallelize them, the preprocessor would have to create a function that can be called by a process. However, this leads to a problem of scoping because the statements may rely on certain variables to exist within its scope that will not be carried over into the newly created function. By limiting parallelization to function calls, and only calling functions written in Python, there is no problem of missing variables. We found this to be a simple and elegant solution. Regardless, however, it is still incumbent upon the programmer to assume full parallelism and adhere to the rules of EFL such as Once Only Assignment to create deterministic parallelized code, even if the code will not actually be parallelized. The programmer should not rely on EFL code to be executed in any particular order.

5 EFL Pre-compiler Implementation

To build the EFL Pre-compiler, we used JavaCC [27], an open source parser generator and lexical analyzer generator. JavaCC generates pure Java code which we compile into a JAR file for distribution. We chose to use JavaCC both for its ease of use and so that the pre-compiler would be compatible with any Java-supported platform. JavaCC works as an *LL(k)* parser [28]. The parser is generated based on a grammar written in EBNF notation, with Java code to handle the processing. This means that any Java debugging tool can be used to help debug a program written with JavaCC, and any classes available in the Java platform can be used in a JavaCC application. Also, by generating and distributing a JAR file, any platform with a Java runtime can run the pre-compiler. This, and having implemented EFL targeting Python as the host language, which also has extensive cross platform support, we ensured that EFL is able to reach as far and wide as possible.

To download the EFL Pre-compiler, see [29].

6 Survey of Previous Work

We are at the beginning of a revolution in programming [30] - a paradigm shift from serial to parallel programming. We have presented EFL, an embedded language that implements some of the principles of "Flexible Algorithms", presented in the companion paper [1]. Here is a brief survey of some of the efforts to deal with this parallel programming revolution.

In [31], the TPL library extensions for parallel programming is described. Using TPL, existing sequential code is potentially parallelizable. In our case, like in openMP [12], it is the programmer's responsibility of determining which parts of the code should be parallelized by using EFL blocks. It is worth to note that in our implementation of EFL, the pre-compiler parallelizes function calls that occur inside EFL-blocks by using Python's parallel programming modules, freeing the programmer from dealing with those Python's modules directly. In contrast to TPL's Parallell.For, in EFL, there are no syntactic difference

between the EFL for-block and its sequential sibling. Iterations of a for block are independent, however the sequence of values of the for index variable are computed sequentially in advance. The code generated using TPL adapts itself to the platform on which it runs, in a similar way to our Python's pools implementation of EFL's parallelism. According to [31], TPL "does not help to correctly synchronize parallel code that uses shared memory". Synchronization problems are avoided in EFL because IN variables cannot be updated from within EFL blocks, and OUT variables are assigned using suitable composite assignments which ensure well defined read values given our scope rules. It is clear that in the non-EFL parts of the program, which are completely sequential, and their semantics are deterministic, the values of variables can be changed without any restriction. Similar difficulties will occur with other libraries which provide parallelism (e.g. MPI [12]), since compile time checks are not possible with library implementations of parallelism.

Etsion et al. [32] describe out-of-order task pipeline execution. Their semantics definition is in terms of sequential execution order, while our semantics definition is in terms of a flexible execution order. They argue that common task-based models require from the programmer to take care of inter-task data dependencies, which is very burdening. To assist the programmer in debugging, we provide a debug mode in which violations of once-only assignments are checked at run-time. For debugged programs, an efficient but insecure compilation mode is provided in which violations of once-only assignment are not checked.

7 Future Work

We see a need for future work in the following areas: (a) Implement EFL for other host programming languages: One of the next major steps with EFL is to implement the pre-compiler for additional host languages. Many programming languages today allow for parallel code, but writing parallelized code in them comes with all of the caveats and dangers of parallel programming. The purpose of EFL is to provide a layer of empathy between the programmer and the native code. Programmers of many languages could benefit from a simple flexible language like EFL; (b) Alternative Execution Modes: Certain decisions were made when writing EFL, and we left many alternative implementation options unexplored; (c) Serial Execution Mode: The intention with EFL, is to provide a language which can be executed flexibly. That means that it can be executed either in serial or in parallel. The main advancement of EFL is the parallel execution, as writing serial code in the host language can of course be accomplished just by using the host language. However, EFL enables the programmer to write deterministic code that retains its determinism in both a serial and parallel execution. Having such an option might also aid the programmer in debugging EFL code; (d) EFL curriculum: By teaching Python along with EFL, students may learn to think both serial and parallel from the beginning of their career. Instead of being taught how to program in a serial manner and then have to "reprogram" their minds to think in parallel, they can understand the intricacies, interactions, and roles of serial and parallel programming. The students will be better equipped to deal with real world problems of application scaling and optimization required for today's applications. In

order to encourage academic institutions to adopt EFL into their curriculum, we would recommend creating a sample curriculum for them to use. The curriculum would have lecture topics, homework assignments, and sample test questions. These materials could be used as guidelines that teachers can adapt to their liking or, if they want, could be used without modification. The curriculum would cover all of the aspects of basic programming. It would teach the fundamentals of algorithms and algorithm costs, teaching the difference between serial and parallel algorithms. In addition, we would also like to establish a website with resources for students learning the language. Resources would include the EFL Pre-compiler available for download, articles on how to use the language, sample programs written using EFL, and a community forum where people can discuss the language. We could record video lectures and instructional videos for teachers and students to walk them through everything from getting started to writing full programs with EFL. Right now the Pre-compiler is called through the command line, but to make the process easier on students we could provide plugins for popular IDEs such as Eclipse and NetBeans.

8 Conclusion

The EFL project's goal was to create an Embedded Flexible Language whose theoretical foundations are based on the order-independent execution approach of "flexible algorithms" (for details, see the companion paper [1]). This kind of language will be embedded into any host language; in its embedded blocks of code, execution can proceed in serial or in parallel order, having deterministic semantics. In this endeavor, we were successful. The EFL language is easy to learn, easy to understand, and has both basic and advanced features that can appeal to first-time programmers and more seasoned ones. This simple, expressive language is capable of powerful hardware utilization unparalleled (pardon the pun) in other languages. The EFL pre-compiler is a robust program that runs in the ubiquitous Java Runtime Environment and its pretty fast at that. We are very pleased with the results and are excited to see what programmers use our pre-compiler for. The EFL language and pre-compiler are tools with exciting potential, and there's no telling how far these tools will go. Just like the inventors of the first serial languages didn't know how much and to what extent their languages will be used, so too EFL is now being given to society with the similar potential as the first languages - but this time with full parallel hardware utilization [33]. Whether it is used in scientific programming, mathematical libraries, graphics processing applications, web programming, as a teaching language to beginning programmers, or even for operating systems, we hope that EFL will advance in usefulness and power wherever it goes.

References

[1] "Flexible Algorithms and their Implementation: Enabling Well Defined Order-Independent Execution", Technical report Jerusalem College of Technology, 2013. Available at http://flexcomp.jct.ac.il/TechnicalReports/Flexalgo&Impl_1_full.pdf

[2] Gordon E. Moore, *Cramming more components onto integrated circuits*, Electronics, Volume 38, Number 8, 1965.

[3] I. Tuimi, *The Lives and Death of Moore's Law*, First Monday, Oct 11, 2002.

[4] A. C. Sodan, J. Machina, A. Deshmeh, K. Macnaughton, and B. Esbaugh, *Parallelism Via Multithreaded and Multicore CPUs*, IEEE Computer, vol. 43, no. 3, March 2010, pp. 24-32.

[5] C.H. Van Berkel, *Multi-Core for Mobile Phones*, invited paper, Design, Automation and Test in Europe DATE'09, March 2009.

[6] S.V. Adve et al. *Parallel Computing Research at Illinois: The UPCRC Agenda*, Nov 2008.

[7] Ami Marowka, *Think parallel: Teaching Parallel Programming Today*, IEEE Distributed Systems Online, Vol. 9, no. 8, 2008.

[8] R. Brown, E. Shoop, J. Adams, C. Clifton, M. Gardner, M. Haupt, and P. Hinsbeeck, *Strategies for Prepapring Computer Science Students for the Multicore World*, ACM ITiCSE-WGR'10, pp. 97-115, 2010.

[9] J. R. Graham, *Integrating Parallel Programming Techniques into Traditional Computer Science Curricula*, inroads - ACM SIGCSE Bulletin, vol. 39, no. 4, pp. 75-78, December 2007.

[10] Patterson, David A. and John L. Hennessy. *Computer Organization and Design*, Second Edition, Morgan Kaufmann Publishers, p. 715. 1998.

[11] Gene M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, Proceedings of the April 18–20, 1967, spring joint computer conference, April 18–20, 1967, Atlantic City, New Jersey

[12] M.J. Sottile, T.G. Mattson, C.E. Rasmussen *Introduction to Concurrency in Programming languages*, Chapman & Hall/CRC, 2010.

[13] M. M. T. Chakravarty, F. W. Schröer, and M. Simons. *V-Nested parallelism in C*. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, Programming Models for Massively Parallel Computers, p. 167–174. IEEE Computer Society, 1995.

[14] A. M. Turing *On Computable Numbers, with an Application to the Entscheidungsproblem*. London Math. Soc. 1938 s2–43: 544–546.

[15] P. Hudak, *Conception, Evolution, and Application of Functional Programming Languages*, ACM Computing Surveys, vol. 21, Nr. 3, p.359–411, September 1989.

[16] Greg Michaelson, *An Introduction to Functional Programming through Lambda Calculus*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1989

[17] Timothy Mattson , Beverly Sanders , Berna Massingill, *Patterns for parallel programming*, Addison-Wesley Professional, 2004

[18] H. Abelson and G. J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd Edition, H. Abelson and G. J. Sussman, 2nd ed, MIT Press, 1996.

[19] K.E. Batcher, *Sorting networks and their applications*, Proceedings of the AFIPS Spring Joint Computer Conference 32, 307–314 (1968).

[20] Leslie G. Valiant, *A bridging model for parallel computation*, Communications of the ACM, v.33 n.8, p.103–111, Aug. 1990

[21] Mark Lutz , Guido Van Rossum, *Programming Python: Object-Oriented Scripting*, O'Reilly & Associates, Inc., Sebastopol, CA, 2001

[22] R.B. Yehezkael, *Flexible Algorithms: Overview of a Beginners* Course, IEEE Distributed Systems Online, vol. 7, no. 11, 2006, art. no. 0611-oy002.

[23] Konrad Hinsen, *Parallel Scripting with Python*, Computing in Science and Engineering,v.9 n.6, p.82–89, November 2007

[24] http://wiki.python.org/moin/Python2orPython3

[25] http://docs.python.org/py3k/library/multiprocessing.html

[26] A. Tucker , A. Gupta, *Process control and scheduling issues for multiprogrammed shared-memory multiprocessors*, Proceedings of the twelfth ACM symposium on Operating systems principles, p.159–166, November 1989

[27] JAVACC Home javacc.java.net

[28] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley,1986.

[29] http://efl.jct.ac.il/

[30] Intel. *Intel Announcement of Parallel Studio project* --- Ami Marowka 2011 IEEE Computer

[31] D. Leijen and J. Hall, *Parallel Performance: Optimized Managed Code For Multi-Core Machines*, MSDN Magazine, Oct. 2007. See http://msdn.microsoft.com/en-us/magazine/cc163340.aspx

[32] Yoav Etsion, Felipe Cabarcas, Alejandro Rico, Alex Ramirez, Rosa M. Badia, Eduard Ayguade, Jesus Labarta, and Mateo Valero, "Task Superscalar: An Out-of-Order Task Pipeline",. *IEEE/ACM Intl. Symp. on Microarchitecture (MICRO-43)*, Dec 2010.

[33] Buyya, R., ed. *Parallel Programming models and paradigms*. High Performance Cluster Computing: Architectures and Systems, Vol. 2. 1999, Prentice Hall PTR: NJ