Multi-core Programming: Implicit Parallelism

Tuukka Haapasalo

April 16, 2009

Outline

Contents

1 Overview 1
  1.1 General Concepts ............................................ 1
  1.2 Examples .................................................. 3

2 Implicit Parallelism 4
  2.1 Futures .................................................. 4
  2.2 Evaluation Strategies ....................................... 6

3 Programming Languages 6
  3.1 Glasgow Parallel Haskell .................................... 6
  3.2 Fortress ................................................... 7
  3.3 Manticore .................................................. 10
  3.4 MultiLisp .................................................. 13

4 References 14

1 Overview

1.1 General Concepts

Comparison of Implicit/Explicit Parallelism

Implicit Parallelism [11]

- Programmer doesn’t define how computation is parallelized
- Compiler parallelizes the execution automatically
- Language’s constructs are inherently parallel
- Often purely functional languages (single-assignment)
- Parallelization not programmed ⇒ no parallelization bugs in code, easier to program

Explicit Parallelism [8]

- Is explicitly defined by the programmer
- Can be difficult to program, debugging is hard
- Examples: threads, OpenMP, MPI, join calculus, data-flow programming, and so on
The separation between explicit and implicit parallelism can be clearly defined in principle: either the programmer defines how the program runs in parallel, or the compiler/interpreter does it automatically, with no input from user. However, the separation in programming languages is not that clear in practice, because the compilers cannot always make implicit parallelism efficient enough on its own. Therefore, some programming languages that support implicit parallelism also contain methods for defining suggestions on how the calculation should be computed in parallel. The suggestions may be given as annotations, defined with futures or by giving evaluation strategies. From this point of view, OpenMP could also be categorized as implicit parallelism, because the programmer does not explicitly control all the aspects of the parallelization.

Examples of Implicit Parallelism

- A pure implicitly parallel programming language can be parallelized with no special directives
- The compiler/interpreter automatically decides which parts of the program are run concurrently
- Parallelization is more straightforward if the language is pure

Calculating the sine of all items in a list [11]

```java
numbers = [0 1 2 3 4 5 6 7];
result = sin(numbers);
```

Pros and Cons of Implicit Parallelism

Pros

- Programmer can concentrate on the algorithms
- Parallel execution separated from the algorithm definition
- Less code required
- Increased programmer productivity

Cons

- No exact control over parallelization
- Parallel efficiency may not be optimal

It is still possible for the user to affect parallelization in some implicitly parallel languages (for example, with evaluation strategies). In fact, Trinder et al. [7] suggest that the best trade-off between ease of programming and efficiency is to use constructs that separate the algorithms from their execution. In their approach, the algorithms are defined on their own, without describing how they should be parallelized (implicit parallelism). In addition, separate parallel execution strategies (not exactly explicit parallelism, but closer) are defined to make the execution efficient.
1.2 Examples

Implicit Parallelism

Examples of languages that support implicit parallelism include:

- **HPF** (High Performance Fortran), an extension to Fortran 90 with constructs that support parallel computing
- **Id**, a general-purpose parallel programming language
- **LabVIEW** (Laboratory Virtual Instrumentation Engineering Workbench), a platform and development environment for visual programming languages, with a graphical language called G.
- **MATLAB M-code**
- **NESL**, a parallel programming language developed at Carnegie Mellon by the SCandAL project
- **SISAL** (Streams and Iteration in a Single Assignment Language), a general-purpose single-assignment functional programming language
- **ZPL** (Z-level Programming Language), an array programming language

The language Id led to the development of the parallel dialect of Haskell, pH.

HPF adds new statements to Fortran for achieving implicit parallelism. These include the ability to create pure procedures.

Conway's Game of Life in ZPL

This excerpt was taken from the ZPL language home page at http://www.cs.washington.edu/research/zpl/comicbook/zcbp3.html:

```zpl
program Life;

config var
  n : integer = 100;
region
  BigR = [0..n+1, 0..n+1];
  R = [1..n, 1..n];
direction
  nw = [-1, -1]; north = [-1, 0]; ne = [-1, 1];
  west = [ 0, -1]; east = [ 0, 1];
  sw = [ 1, -1]; south = [ 1, 0]; se = [ 1, 1];
var
  TW : [BigR] boolean; -- The World
  NN : [R] integer; -- Number of Neighbors

procedure Life();
begin
  -- Initialize the world
  [R] repeat
    -- Count live neighbours
    NN := TW@nw + TW@north + TW@ne +
      TW@west + TW@east +
      TW@sw + TW@south + TW@se;
    -- Update the world
    TW := (TW & NN = 2) | (NN = 3);
    until !(|<< TW);
end;
```
In ZPL, the operator \( |<< \) is a logical OR, so the comparison \( !(|<< TW) \) is true only when none of the boolean variables in the world (TW) are true.

2 Implicit Parallelism

Techniques for Implicit Parallelism

- Futures
- Evaluation Strategies
- Parallel Structures, Arrays
- Annotations
- Methods and Function

2.1 Futures

- Constructs used for synchronization
- Refer to objects whose value is not initially known
- Futures can be passed around in code like normal variables
- Synchronization occurs when the value of the future is specifically requested
- Futures are inherently pure (single-assignment)

An example:

```lisp
a = future do-calculation;
b = future do-other-calculation;
... c = a + b;
```

In the example, the values of \( a \) and \( b \) are only needed at the last line, and thus might not be evaluated before that. Basically, the evaluation of \( a \) at the last line will block until the value of \( a \) is available, but of course the computation of \( b \) may occur in parallel while waiting for the value of \( a \).

Future Terminology

- *Delays* and *futures* seem to be used interchangeably
  - In MultiLisp: delay calculation not started before it is needed
- A *promise* is more ambiguous, but one definition is a a single-assignment variable which may be set by any thread
  - Usually can be set only once
  - Reading a promise before the value has been set creates a future for the value
- Future is *implicit* if it is used like a normal variable, *explicit* if there is a special function for explicitly fetching the value
The use of futures is classified in Wikipedia [10] as implicit, when the future is used as if it were an ordinary reference; and explicit, when the value of the future is specifically requested by calling a function that forces the value. By this definition, implicit futures require support from the host programming language, and explicit futures can be implemented as a library service to a language that has no such support.

**Futures in Programming Languages**

Futures have been implemented in the following languages:

- Id
- MultiLisp
- Java (explicit futures only)
- Scheme
- C++0x (explicit futures only)
- Alice ML, Io, Oz, Lucid, AmbientTalk, R, …

C++0x is a planned revision of the C++ standard, designed to replace the existing standard, with the most recent draft published in March 2009. The standard is planned to be published this year, thus changing the name of the standard to C++09.

**Future Examples**

**Futures in Java**

```java
void showSearch(final String target) throws InterruptedException {
    Future<String> future = executor.submit(new Callable<String>() {
        public String call() {
            return searcher.search(target);
        }
    });
    displayOtherThings(); // do other things while searching
    try {
        displayText(future.get()); // use future
    } catch (ExecutionException ex) { cleanup(); return; }
}
```

The invocation `future.get()` explicitly requests the program to block until the future’s value is available.

The Java example contains (taken from Java 1.6 API documentation) contains an executor framework with other components not show in this excerpt. A more simple example is just to create an instance of `FutureTask`, which implements the `Future` interface, and give the `Callable` subclass as a parameter to it. The default implementation of `FutureTask` blocks when calling the `get()` method until the `Callable` object given as a parameter has finished (or until the task has been cancelled).
Future Examples

Futures in Alice ML

```ml
val x = spawn fib n;
```

In Alice ML, the computation will block when the value `x` is required.

Futures in MultiLisp

```lisp
(cons (FUTURE a) (FUTURE b))
```

In this example, the computation of `a`, `b` and the `cons` construct will be overlapped until the value of `a` or `b` is actually needed.

2.2 Evaluation Strategies

- Is it enough just to write the functional specification?
- Unfortunately, not quite so straight-forward
- Some hints on how the computation should be parallelized are still needed

Evaluation Strategies

An evaluation strategy is a function that specifies the dynamic behaviour of an algorithmic function. [7]

- Thus, an efficient parallel program = algorithms (functional specification) + evaluation strategy
- Benefit: a clear separation of the algorithm from the parallel processing coordination

Evaluation strategies in the article by Trinder et al. are designed for Glasgow Parallel Haskell (GpH), which is an extension of Haskell.

3 Programming Languages

Now for the programming language examples...

3.1 Glasgow Parallel Haskell

- Based on the Glasgow Haskell Compiler (GHC)
  - GpH forked from GHC some ten years ago
- Purely functional programming language
- [http://www.macs.hw.ac.uk/~dsg/gph/](http://www.macs.hw.ac.uk/~dsg/gph/)
- Not updated for a while
- Current version (even the stable one) does not compile, and the installation instructions are outdated. Perhaps could work on another Linux distribution/version?

GHC is perhaps the most commonly used Haskell compiler, and is being actively developed even now. GHC itself now supports explicit parallelism with commands for forking and killing threads, synchronizing variables and so on. It seems that this project is dead now. The latest activity reports are from 2005.
Evaluation Strategies in Parallel Haskell

Parallelism is introduced in GpH with the keywords:

- ‘par’ takes two arguments to be evaluated in parallel. p 'par' e has the value of e, but p is computed in parallel (lazily, like a future).

- ‘seq’ sequentializes the computation. p 'seq' e evaluates p before returning with the value e.

- ‘using’ makes an algorithm use a given evaluation strategy.

General-purpose evaluation strategies can be defined to form a library that can be taken into use by simply placing ‘using <strategy>’ into the computational algorithm definition. In practice, more specific evaluation strategies may need to be defined to get the best performance.

Sequential quicksort

```haskell
main = print (quicksort ([999,998..0]::[Int]))
quicksort [] = []
quicksort [x] = [x]
quicksort (x:xs) = (lo ++ (x:hi))
where
  lo = quicksort [ y | y <- xs, y < x]
  hi = quicksort [ y | y <- xs, y >= x]
```

Parallel Quicksort

```haskell
import Strategies
main = print (quicksort ([999,998..0]::[Int]))
quicksort [] = []
quicksort [x] = [x]
quicksort (x:xs) = (lo ++ (x:hi)) 'using' strategy
where
  lo = quicksort [ y | y <- xs, y < x]
  hi = quicksort [ y | y <- xs, y >= x]
  strategy result = rnf lo 'par'
  rnf hi 'par'
  rnf result
```

The strategy rnf (reduce to normal form) forces the sublists to be evaluated before returning the result back up the call stack. The strategy is embedded to the algorithm definition here, but it could be presented separately also.

3.2 Fortress

Quick facts [9, 6]

- New programming language, initially drafted by Sun Microsystems
- Intended as a successor for Fortran (but not similar to)
- Not tied to legacy language syntax or semantics
- Not pure
• Features include abstraction, type safety and full Unicode support
• Concrete syntax that is similar to mathematical notation
• Source code can be rendered as ASCII text, Unicode text or as a prettied image (can also be converted to LaTeX)

More on Fortress

Practicalities
• Runs on top of Java (JVM) 1.6
• Should be cross-platform
• None of the tested distributions compiled in my PC
• Binary distribution (build 3625) works!
• http://projectfortress.sun.com/Projects/Community/wiki

Implicit Parallelism [2]
• LIFO work queue
• Potential parallelism is split into tasks that are fed to the work queue
• Any thread that is free takes a task from the queue (work stealing)

Sun’s Fortress interpreter runs on top of a JVM and requires Java 1.6 to run. This project seems to be alive and in development. From practical viewpoints, the Fortress interpreter/compiler is not yet finalized (compiler part can only compile simple programs, and so on), and the documentations are scarce. However, with Sun backing the project, there is hope that this language can actually become something.

Fortress details
Some details on the Fortress language [9, 2, 6]:
• Syntax similar to Scala, Standard ML, and Haskell
• Designed for high-performance computing (HPC)
• Potential parallelism is the default (automatically tuned to available hardware)
• Parallelism achieved with the LIFO work queue
• Threads split whenever possible

Example
[: 2 * n | n in nums where n > 0 :]

The threads split whenever they notice some possible parallelism. They generally try to split in half, moving half of the computation to the work queue, and continuing themselves on the rest of the work. Thus, any other idle thread may directly start working on the other half of the work. The LIFO queue guarantees that the oldest tasks will be started first, which keeps the response times low.
Implicitly parallelized constructs

The list shown below lists the different constructs that are automatically parallelized in Fortress [3]:

- Tuple expressions: \( (e_1, e_2, e_3) \)
- also do blocks
- Method invocations, function calls
- for loops, comprehensions, sums, generated expressions, big operators
- Extremum expressions
- Tests

Extremum expressions find the most extreme (largest, smallest, etc.) branch of a case expression:

```fortress
case most> of
    1 mile => "miles are larger"
    1 kilometer => "we were wrong again"
end
```

Tests are top-level functions annotated with the keyword `test`. They work as unit tests for the program, and have some special functionality (non-tests may not call tests, all tests can be run with a single command).

Code Rendering Examples

The following example is taken from the Fortress distribution package. The Fortress code is in `Fortify/example/buffons.fss`.

Fortress Code in ASCII

```fortress
for i <- 1#3000 do
    delta_X = random(2.0) - 1
    delta_Y = random(2.0) - 1
    rsq = delta_X^2 + delta_Y^2
    if 0 < rsq < 1 then
        y1 = tableHeight random(1.0)
        y2 = y1 + nLen(delta_Y / sqrt(rsq))
        (y_L, y_H) = (y1 MIN y2, y1 MAX y2)
        if ceiling(y_L/nLen) = floor(y_H/nLen) then
            atomic do hits += 1.0 end
        end
        atomic do n += 1.0 end
    end
end
```

Rendered Fortress Code

```fortress
for i ← 1#3000 do
    Δ_X = random(2.0) - 1
    Δ_Y = random(2.0) - 1
    rsq = Δ_X^2 + Δ_Y^2
    if 0 < rsq < 1 then
        y1 = tableHeight random(1.0)
        y2 = y1 + nLen(Δ_Y / sqrt(rsq))
        (y_L, y_H) = (y1 MIN y2, y1 MAX y2)
        if ceiling(y_L/nLen) = floor(y_H/nLen) then
            atomic do hits += 1.0 end
        end
        atomic do n += 1.0 end
    end
end
```
(y_L, y_H) = (y_1 \text{MIN}, y_1 \text{MAX}, y_2)
if ceiling(y_L / nLen) = floor(y_H / nLen) then
   atomic do hits += 1.0 end
end
atomic do n += 1.0 end
end

In Fortress, the \texttt{for} loop is automatically parallelized (by the default implementation of \texttt{for}) [9]. That is, the loop in the example might not necessarily run sequentially in a strictly linear order, but it can be run in parallel depending on the underlying hardware. The \texttt{for} is, however, just a library function that can be replaced by another implementation if the programmer so wishes.

**Buffon’s Needle**

\( nLen = 20 \)
\( numRows = 10 \)
\( tableHeight = nLen \times numRows \)

\( \text{run(args : String ...):() = do} \)
\( \text{var hits : F64 := 0.0} \)
\( \text{var n : F64 := 0.0} \)
\( \text{for } i \leftarrow 1 \# 3000 \text{ do} \)
\( \delta_X = \text{random}(2.0) - 1 \)
\( \delta_Y = \text{random}(2.0) - 1 \)
\( rsq = \delta_X^2 + \delta_Y^2 \)
if \( 0 < rsq < 1 \) then
\( y_1 = tableHeight \times \text{random}(1.0) \)
\( y_2 = y_1 + nLen(\delta_Y / \text{sqrt}(rsq)) \)
\( (y_L, y_H) = (y_1 \text{MIN}, y_1 \text{MAX}, y_2) \)
if ceiling(y_L / nLen) = floor(y_H / nLen) then
   atomic do hits += 1.0 end
end
atomic do n += 1.0 end
end
probability = hits/n
\( \pi_{est} = 2.0 / \text{probability} \)
print("estimated Pi = ")
println(\( \pi_{est} \))
end

The example given here calculates an estimation of \( \pi \). The example is based on the problem called Buffon’s Needle: if you have a needle of length \( n \), and a paper that has perpendicular lines drawn on it that are spaced exactly \( n \) apart from each other, then the possibility for the needle crossing a line when it is dropped on the paper is \( 2/\pi \). This is an example of an embarrassingly parallel computation, except for the part of updating the hit counts.

### 3.3 Manticore

- New language, work started in 2007 [5]
- Heterogeneous parallel programming language
• Multiple levels of parallelism (explicit parallelism, data-parallel arrays)

• Coarse-grained, explicit parallelism based on Concurrent ML (CML)

• Fine-grained nested parallelism based on data-parallel languages (NESL, Nepal)

• Implicit threading based on annotations

• Around a dozen papers published on the language

• No implementation published yet

The explicit concurrency in Manticore is implemented with manual thread definitions and synchronous message passing. Implicit concurrency in Manticore includes implicit threading, where annotations provided by the programmer are used to parallelize computation automatically. [1]

From practical point-of-view, the authors have been very active publishing the papers, so there is no lack of activity in this project. The lack of a published interpreter/compiler is therefore the only current disadvantage of this language...

Implicitly-threaded Parallelism

• Hints to the compiler on what could be parallelized

• Sequential semantics
  
  – Deterministic model
  
  – Care required to ensure correct parallelized computation ordering with message passing and exception handling

• Parallel arrays, parallel tuples, parallel bindings, and parallel cases

Parallel arrays

• \([e_1, e_2, \ldots, e_n]\) creates a parallel array with members \(e_1, e_2, \ldots, e_n\)

• \([| 2 * n | n \text{ in } \text{nums} \text{ where } n > 0 |]\) creates a parallel comprehension

• Hints that the different elements can be calculated in parallel

Comprehensions can be used to specify both SIMD parallelism (mapped onto vector hardware) and SPMD parallelism (mapped onto multiple cores). If an exception occurs when a computing a given index \(i\), the computation must wait until the previous elements with indexes \(< i\) have been calculated before throwing the exception. [1]

Implicitly-threaded Parallelism

Parallel tuples

• \((e_1, e_2, \ldots, e_n)\) creates a parallel tuple with members \(e_1, e_2, \ldots, e_n\)

• The semantics for the tuple are fork-join

• The result is a normal value

• Different parts can have different types
Parallel bindings

- More flexible scheduling
- pval $p = e$ launches the evaluation of $e$ in a parallel thread
- Lazy evaluation, so this is a future
- If the value is never used, the calculation is canceled

With parallel tuples, the separate elements are calculated in parallel, joining the threads before moving on in the computations. The following code example shows how to calculate a product of the leaves of a binary tree in parallel.

```ml
datatype tree = Lf of int | Nd of tree * tree

fun trProd (Lf i) = i
| trProd (Nd (tL, tR)) = op* (|trProd1 tL, trProd1 tR|)
```

The parallel bindings work like futures. The following code example shows the same leaf product calculation using parallel bindings with short-circuiting (a found zero prunes away unnecessary calculations). The pruned branches are not calculated because their value is never requested (after the $pL = 0$ computation the program knows whether $pR$ is needed or not).

```ml
fun trProd (Lf i) = i
| trProd (Nd (tL, tR)) = let
  pval pL = trProd2 tL
  pval pR = trProd2 tR
  in
  if (pL = 0) then 0
  else (pL * pR)
  end
```

Parallel cases

- The separate discriminants are evaluated in parallel, and any matching one is taken (first one that matches, possibly)
- Non-deterministic selection if multiple matching branches
- A variant is the parallel choice, val $a = b |?| c$

The parallel choice $e1 |?| e2$ is syntactic sugar to the following:

```ml
pcase e1 & e2
  of x & ? => x
  | ? & x => x
```

An example with the same tree product calculation with parallel case is shown below. Whichever of the branches match first is used to calculate the value. For the parallel choice, the first one of $b$ and $c$ to obtain a value will define the value of $a$.

```ml
fun trProd (Lf i) = i
| trProd (Nd (tL, tR)) = (pcase trProd(tL) & trProd(tR)
  of 0 & ? => 0
  | ? & 0 => 0
  | pL & pR => pL * pR)
```
Manticore Example

Parallel Alpha-Beta Pruning [1]

fun maxT (board, alpha, beta) =
  if gameOver(board) then
    Rose ((board, boardScore board), [||])
  else let
    val ss = successors (board, X)
    val t0 = minT (ss!0, alpha, beta)
    val alpha’ = max (alpha, treeScore t0)
    fun loop i = 
      if (i = (plen ss)) then [||]
      else let
        val ts = loop (i+1)
        val ti = minT (ss!i, alpha’, beta)
        in
          if (treeScore ti) >= beta then
            [|ti|] (* prune *)
          else
            [|ti|] |@| ts
        end
      end
    val ch = [|t0|] |@| loop(1)
    val maxScore = maxP [| treeScore t | t in ch |]
  in
    Rose ((board, maxScore), ch)
  end

3.4 MultiLisp

- MultiLisp is a dialect of Scheme that is extended to support parallel execution [12]
- Like Scheme: lexical (static) scoping, tail recursion
- Not pure, some constructs cause side effect
- Also includes explicit parallelism
- Not a new idea, the MultiLisp article [4] is from 1985
- MultiLisp was implemented in Interlisp, which has now been dead for some 20 years

The MultiLisp article [4] categorizes futures and parallel calls (see next slide) as explicit parallelism. This again shows that the separation of implicit/explicit parallelism is not that clear. This is purely a theoretical curiosity now, because the only interpreter (Interlisp) has been dead for so long, and no other Lisp interpreter has implemented MultiLisp.

- Parallel execution: (PCALL Fun A B C ... ) is equivalent to (Fun A B C ...), except that A B C are executed in parallel (not sequentially left-to-right)
- Futures: (cons (FUTURE a) (FUTURE b)) overlaps the execution of a, b and the cons construct until the value of a or b is actually needed.
• Delays: \((\text{cons } \alpha (\text{delay } \beta))\) constructs a pair whose second value is calculated only when needed (calculation not started in advance).

In the example below, if \(\text{X}\) is \texttt{future}, the calculation may start immediately. If \(\text{X}\) is \texttt{delay}, the calculation only starts when that element of the list is requested.

\[
\text{(defun ints-from } (n) \\
\text{ (cons } n \\
\text{ (X (ints-from (+ n 1))))})
\]

Parallel Quicksort in MultiLisp [4]

\[
\text{(defun qsort } (l) \text{ (qs } l \text{ nil))} \]
\[
\text{(defun qs } (l \text{ rest}) \text{ (if (null } l) \\
\text{ rest) \text{ (let ((parts (partition (car l) (cdr l))))} \\
\text{(qs (left-part parts) \\
\text{ (future (cons (car l) (qs (right-part parts) rest))))))})}
\]

\[
\text{(defun partition } (sep } l\text{) \text{ (if (null } l) \\
\text{ (bundle-parts nil nil) \\
\text{ (let ((cdrparts (future partition sep (cdr lst))))} \\
\text{ (if (> sep (car lst)) \\
\text{ (bundle-parts (cons (car lst) \\
\text{ (future (left-part cdrparts)))} \\
\text{ (future (right-part cdrparts)))} \\
\text{ (bundle-parts (future (left-part cdrparts))} \\
\text{ (cons (car lst) \\
\text{ (future (right-part cdrparts)))))})}))
\]

\[
\text{(defun bundle-parts } (x \text{ y) (cons } x \text{ y))} \]
\[
\text{(defun left-part } (p) \text{ (car } p) \\
\text{(defun right-part } (p) \text{ (cdr } p))
\]

4 References

References


