Bulletin of the *Transilvania* University of Braşov • Vol 6(55), No. 2 - 2013
Series III: Mathematics, Informatics, Physics, 61-72

UPON THE PERFORMANCE OF A HASKELL PARALLEL IMPLEMENTATION

Alexandra BĂICOIANU[1], Raluca PÂNDARU[2] and Anca VASILESCU [3]

Abstract

The `Haskell` developers focus on providing an open range of packages and libraries in various research areas. Particularly, image processing is naturally expressed in terms of parallel array operations and we use here `Repa` as a great tool for coding image manipulation algorithms.

Our target is to analyze the execution time of a `Haskell` parallel implementation and also to compare the results to the appropriate `C++` implementation. A certain example from the image processing area of interests is selected. The conclusion is that the compared execution time values depend both on the physic and the logic parameters of the applied solutions.

2000 *Mathematics Subject Classification:* 68N18, 68U10, 68U99, 65Y05.
Key words: parallel programming, `Haskell`, `Repa`, `C++`, image processing, edge detection.

1 Introduction

Nowadays, `Haskell` is accepted to be an advanced purely-functional programming language. As an open-source product of more than twenty years of cutting-edge research, it allows rapid development of robust, concise, correct software. With strong support for integration with other languages, built-in concurrency and parallelism, debuggers, profilers, rich libraries and an active community, `Haskell` makes it easier to produce flexible, maintainable, high-quality software. [1]

Our results are based on analyzing the execution time of a `Haskell` parallel implementation and also on comparing the results to the appropriate `C++` implementation. The example has been chosen from image processing area of interests

[1]Faculty of Mathematics and Computer Science, *Transilvania* University of Braşov, Romania, e-mail: a.baicoianu@unitbv.ro

[2]Faculty of Mathematics and Computer Science, *Transilvania* University of Braşov, Romania, e-mail: raluca_pandaru@yahoo.com

[3]Corresponding author - Faculty of Mathematics and Computer Science, *Transilvania* University of Braşov, Romania, e-mail: vasilex@unitbv.ro

and the purpose is to detect the edges on a given image. The basic scenario is following the next steps: the first step transforms the color image into a gray-scale one, the second step applies a *Gaussian filter* on the image and after that, the final step uses the *Sobel operator* for the edge detection. This solution is appropriate for analyzing how `Haskell` language works in this context. The comparison to the `C++` similar results is useful, taking into consideration that `C++` is considered as being one of the best languages for image processing.

Following the `Haskell` support, `Repa` is a great tool for coding image manipulation algorithms, which tend to be naturally parallel and to allow a big amount of data. In [1] one may find as example a program for rotating an image around its center by a specified number of degrees. In this paper, we use the `Repa` library for developing a program for edge detection using the *Sobel operator*. The [4] refers the authors' contributions to the `Repa` extensions for achieving the performance comparable to the standard `OpenCV` library, providing modern parallel implementations for a set of image processing algorithms.

In our work, it is an advantage to adopt one of the `Haskell` valuable conventions, namely working with curried functions, the function type is assumed to associate to the right. On the other hand, we note that in the current step of an expression evaluation, the reductions may be performed simultaneously on many parts of that expression. It yields that a function evaluation might be done using many collaborative processes, by involving parallel and concurrent processes. This solution leads to consider the functional programming languages, in general, and particularly the `Haskell` language as a pure functional programming language appropriate for dual-core or even quad-core processing models and architectures.

An important contribution of the authors in this paper consists in obtaining a parallel `Haskell` implementation of an edge detection algorithm based on the *Sobel operator*, beyond the existing `Repa` examples. The original part comes from the solution proposed for measuring the running time performances on the specified `Haskell` codes. As an overall conclusion, we may say that `Haskell` is an appropriate alternative for developing the image processing algorithms, especially if we could have support for applying a 7x7 or even bigger masks for corresponding *Gaussian filters*.

2 Theoretical aspects

This section provides an overview of some aspects which are relevant for understanding the main issues of the paper: the edge detection algorithm and the parallel approach for software development.

2.1 Edge detection

In this section, the main ideas that underlie the specific image transformations are presented, following the three steps: (1) image transformation from a color image into a grayscale one, (2) applying a *Gaussian filter* and finally (3) applying the *Sobel operator* for completing the edge detection solution.

A picture is represented like a 2D quadratic matrix of pixels. For a color image coding, every pixel has a color. Computers use 3 different colors, namely red, green and blue and each pixel color means a specific combination of these three basic colors. An integer value between 0 and 255 is used for representing how strong the corresponding color is in the composition. Usually, in image processing the gray-scale representation is used. So, those 3 color-values associated for a pixel are replaced with a gray tone. To calculate this gray tone, a specific average is used, following the formula (1).

$$\left.\begin{array}{l} pixel(R,G,B) \in color\ image \\ pixel(V) \in grayscale\ image \end{array}\right| \Rightarrow \begin{cases} V = \frac{R+G+B}{3} \\ or \\ V = 0.3R + 0.59G + 0.11B \end{cases} \tag{1}$$

After we have applied this transformation to all of the involved pixels, we have the picture represented as a surface with many landforms (e.g. plans, hills, mountains), where a landform height is depending on the pixel's values considered in the area of that form.

For the second step, we mention that in the image processing context, filters are mainly used to suppress either the high frequencies in the image, i.e. smoothing the image, or the low frequencies, i.e. enhancing or detecting edges in the image. The filter is described by an $n \times m$ discrete convolution mask - an array or a matrix with constant values. In particular, the *Gaussian filter* is a widely-used effect in image processing, typically to reduce image noise or unwanted details. The visual effect of this blurring technique is a smooth blur, resembling it by viewing the image through a translucent screen or producing it by the shadow of an object under less illumination. The *Gaussian filter* is a low-pass filter, attenuating high frequency signals.

In order to detect the image edges in step 3, we also have to set exactly what an edge is, in terms of image processing. Intuitively, we can make a distinction between an object and another because there is a big enough contrast, meaning a color/tone difference, between them. As the contrast is lower, it is more difficult to distinguish objects from each other. In a specific context, an edge can be seen as a sufficiently large difference between two specific values of neighboring pixels. From the mathematical point of view, these differences can be evaluated using the partial derivative (note that the function has two variables). Geometrically speaking, this partial derivative defines the slope of the tangent to the graphic in that point. The bigger the difference, the larger the slope of the tangent is.

Following the ideas from [9], "A number of edge detectors based on a single derivative have been developed by various researchers. Amongst them, the most important operators are the Robert operator, *Sobel operator*, Prewitt operator, Canny operator, Krisch operator, etc. In each of these operator-based edge detection strategies, we compute the gradient magnitude in accordance with a specific formula. If the magnitude of the gradient is higher than a threshold, then we detect the presence of an edge."

2.2 Parallel programming

Parallel programming refers the technique which allows software applications to use the hardware resources to the fullest. We mean that if a PC has two cores, then certain code fragments will be executed by one of the cores and others by the other core, thus they are being executed at the same time. This way, the execution time is reduced, ideally to half, comparing to the situation of the whole code executed on one core. As it can be inferred, unexpected results may occur if the code is not written properly. This is, in fact, the challenge which faces the compilers and the libraries which support parallel programming.

A parallel program is one that uses a multiplicity of computational hardware (e.g., several processor cores) to perform a computation more quickly. The aim is to arrive at the answer earlier, by delegating different parts of the computation to different processors that execute at the same time. For parallel programming, we would like to use deterministic programming models if at all possible. Since the goal is just to arrive at the answer more quickly, we may count on the modern processors architecture based on implementing the deterministic parallelism in the form of pipelining and multiple execution units. [1]

2.2.1 Haskell support

An important support for `Haskell` developers is provided by the `GHC` (*Glasgow Haskell Compiler*) which implements the major `Haskell` extensions in order to facilitate concurrent and parallel programming. Parallelism means running a `Haskell` program on multiple processors, with the goal of improving performance. Ideally, this should be done invisibly, and with no semantic changes. Concurrency means implementing a program by using multiple I/O-performing threads. While a concurrent `Haskell` program can run on a parallel machine, the primary goal of using concurrency is not to gain performance, but that is the simplest and most direct way to write the program. Since the threads perform I/O, the semantics of the program is necessarily non-deterministic. `GHC` supports both concurrency and parallelism. [13]

In [1] at least two methods for parallel programming in `Haskell` there are specified:

- The `Eval` Monad, for basic operations;
- The `Par` Monad, to ensure the parallelism of the data flow.

Besides these methods, we can also use the `Repa` library and a set of specific packages with prefix-name `Repa` (*REgular PArallel arrays*), for example [14]:

- `Repa` for working with multidimensional vectors that supports parallel execution
- `Repa-io` for reading and writing vectors in different formats, including `BMP`
- `Repa-algorithms` proposing several algorithms for the `Repa` package, which can be reused also in other contexts

Following the [1], `Repa` library provides a rich set of combinators for building parallel array computations. You can express a complex array algorithm as the composition of several simpler operations, and the library automatically optimizes the composition into a single-pass algorithm using a technique called *fusion*. Furthermore, the implementation of the library automatically parallelizes the operation using the available processors.

`Repa` is also used to implement the `DPH` (*Data Parallel Haskell*) [12] library, another option for parallel programming, which was added in `GHC 7.4`. Both libraries, `Repa` and `DPH` can be used for parallel execution, but only on multiple cores architectures, since distributed and parallel executions are not yet supported by the compiler. The difference between the previous libraries is that `Repa` is used for working with regular data structures, while `DPH` works with irregular data structures.

Just because `Repa` allows to write and read data in `BMP` format using the `Repa-io` package, this library represents an appropriate choice for solving the parallel proposed issues. The library also provides another package called `Repa-devil` for reading and writing the images in different formats. In order to use this package, the `DevIL` library is required to be installed. This is a library written in `C++`, suitable for reading and writing files. The package `Repa-devil` is just a wrapper over it, allowing the `GHC` compiler to communicate with the code written in `C++`. Hence, we can work without this last package, because `Repa` can read and write images in `BMP` format, avoiding the link with another external library. It is true that this decision limits the work with images, but even in the case of those images in `BMP` format, it is required that these should be the 24-bits Bitmap images. This shortcoming can be easily overcome, because image-tools allow the conversion to `BMP` format.

In order to run in parallel a program which is developed with `Repa`, it is necessary to specify the following arguments: `+RTS -N<nr_core-uri>` or `+RTS -N`. In the second case, the `GHC` compiler has to determine by itself how many cores exist on the local machine. If you specify more or fewer cores, you may notice a significant increase in execution time.

For obtaining a good performance, it is not enough to indicate just that the execution is done on many cores. It is also necessary to make some optimizations. For example, when you compile the code to a standalone executable, you must add a parameter that will indicate what level of optimization the compiler will apply. Such a command would be: `ghc -O2 <fisier.hs> -threaded`. Leaving the parameter `-O2` out will greatly reduce compilation time because the compiler will not do any optimization. In addition to this, regarding the code, functions that work with vector types from the `Repa` library, have to be marked as inline, using `{-# INLINE <nume_functie> -#}`.

3 Developed applications

In this section we present the `Haskell` application previously described and we explain how we have applied the previous three steps from section 2.1 for developing this implementation.

In order to use the `Repa` library, the packages introduced in section 2.2.1 must be installed. For `Haskell 2012.4.0.0` platform which uses `GHC 7.4`, this installation is done using the generic command `cabal install`, like that:

```
cabal install repa-3.2.1.1
cabal install bmp-1.2.1.1
cabal install repa-io-3.2.1.1
cabal install repa-algorithms-3.2.1.1
```

As you may see, the version of each package installed has to be explicitly specified. This is important because the `cabal` command always installs the newest version, if none is specified. The actual version of the compiler is `GHC 7.6`, while the last `Haskell` platform uses `GHC 7.4` and the newest versions of the `Repa` packages are using `GHC 7.6`. So, by installing the last versions, conflicts may appear between platform packages and the ones that `Repa` library needs.

The second command from the ones above, installs the `bmp` package to avoid the use of another version chosen by `cabal`. For selecting a compatible version with the `Haskell` platform, one may follow the specifications from [14] for the given packages `Repa`, `Repa-io` and `Repa-algorithms`. For running the compiled version of the program the `Repa` library is not needed.

Next we will present some bits of code used to implement the steps from section 2.1.

3.1 Step 1. Image Transformation: color to grayscale

For converting a 24-bits Bitmap image into a single channel image, the following operations have to be done:

- adding references:

  ```
  import Data.Array.Repa.IO.BMP
  import Data.Array.Repa as R
  import GHC.Word as W
  ```

- defining two types of synonym data:

  ```
  type ImageC3 = Array U DIM2 (W.Word8, W.Word8, W.Word8)
  type ImageC1 = Array U DIM2 Float
  ```

  The first synonym type `ImageC3` is defined for an RGB image with 3 channels and the second one, `ImageC1`, is for a single-channel image. Following the `Repa` content, an array definition has the syntax: `Array r DIM<dimension_nr>`, where `r` represents the vector type and `a` specifies the element type. In this paper only the vector types `U` - unboxed and `D` - delay are used, but these are not the only two available types in `Haskell` [1]. In the previous definitions `U` comes from an unboxed array. The `Haskell` unboxed arrays are very similar with the ones from `C++`, because their indexing is done very fast, without using multiple redirects. Besides, they are parsed at declaration stage, so that they do not benefit from the lazy evaluation advantages from Haskell [12]. The parameter `DIM2` specifies that the vector will have two dimensions, so that the image can be seen in 2D.

- reading the image in the `main` method, using `readImageFromBMP`; depending on the returned result, the program will continue or not. In order to get the image path, the function `readImagePath` is used.

```
 main :: IO
 main = do
    inImagePath <- readImagePath "Path of inImage: "
    result <- readImageFromBMP inImagePath
    case result of
       Left bmpError -> putStrLn (show bmpError)
       Right inImage -> run inImage
```

- If the image has successfully read, then the function `run` is called with an image-type `ImageC3` as parameter. Its role is to collect all the needed parameters: final image path, *Gaussian filter* size and the parameter for the *Sobel operator*. After that, the conversion from `ImageC3` to `ImageC1` will be done by applying the function `toImageC3` and the edge detection algorithm is called.

```
run :: ImageC3 -> IO ()
run inImage = do
    outPath <- readImagePath "Path of outImage: "
    sobelThreshold <- readFloatValue "Sobel threshold([0, 1)):"
    gaussianKernelSize <- readGaussianKernelSize
    putStrLn "Converting image to gray scale..."
    imageC1 <- toImageC1 inImage
    putStrLn "Performing edge detection algorithm..."
    (resultImageC1, elapedTime) <-
        time (applyEdgeDetection imageC1
                    gaussianKernelSize sobelThreshold)
    ...
```

  Note that we are starting now the execution-time evaluation using the function `time`. For this, a new reference is needed, as follows:

```
import Data.Array.Repa.IO.Timing
```

  Function `toImageC1` is described below. It is an `inline` function and it uses `computeP` and `R.map` for working in parallel. Function `floatLuminanceOfRGB8` is defined in the `Data.Array.Repa.IO.BMP` and it has to convert the elements from the tuple-type `(W.Word8 W.Word8 W.Word8)` into `Float`, corresponding to the transformation of the color image into a gray scale image.

```
toImageC1 :: ImageC3 -> IO ImageC1
{-# INLINE toImageC1 #-}
toImageC1 imageC3=computeP (R.map floatLuminanceOfRGB8 imageC3)
```

3.2 Step 2. Applying a *Gaussian filter*

Next step consists in applying the *Gaussian filter*. This is done in the function named `applyEdgeDetection` and listed below.

As you may see, the function `applyEdgeDetection` receives the image and the size of the *Gaussian filter* as parameters. Its role is to assure that the mask with the indicated size is going to be applied. Note that only 3 dimensions can be specified for the `gaussianKernelSize` following the definitions established by the module

`Data.Array.Repa.Stencil.Dim2`. Important and useful details about the Haskell implementations of specific image processing concepts are presented in [4], [7].

```
applyEdgeDetection :: ImageC1 -> Int -> Float -> IO DelayedImageC1
applyEdgeDetection imageC1 gaussianKernelSize sobelThreshold = do
    applyGaussianFilter imageC1 gaussianKernelSize
              >>= applySobel sobelThreshold
              >>= return

applyGaussianFilter :: ImageC1 -> Int -> IO ImageC1
applyGaussianFilter imageC1 gaussianKernelSize =
      case gaussianKernelSize of
           3 -> applyGaussianFilter3 imageC1
           5 -> applyGaussianFilter5 imageC1
           7 -> applyGaussianFilter7 imageC1
```

The algorithm for apllying a mask on an image is implemented as above and this requires new references, such as:

```
import Data.Array.Repa.Stencil as A
import Data.Array.Repa.Stencil.Dim2 as A
```

All of the three functions called by `applyGaussianFilter` are almost the same, the only difference being given by the form of the mask. So, we present here only one of them:

```
applyGaussianFilter3 :: ImageC1 -> IO ImageC1
{-# INLINE applyGaussianFilter3 #-}
applyGaussianFilter3 image = computeP (
                         R.map (/ 15) (forStencil2 BoundClamp image
                               [stencil2| 1 2 1
                                          2 3 2
                                          1 2 1 |]))
```

Function `forStencil2` applies the mask on the image received as the second parameter. The first parameter specifies the strategy used for those pixels the mask cannot be applied on (the pixels from the image edge). In this case, the `BoundClamp` value specifies that the edge pixels will be replaced by the specific value calculated using the existing pixels' neighbors. Then, the last parameter specifies in a list the form and the coefficients of the mask. These coefficients are not normalized, so it follows that the function `R.smap` will divide each pixel value by 15.

In order to have an appropriate mask definition two arguments have to be specified to the compiler. These might be included into the top of the source code, or they might be sent through a command line. We chose the first solution by introducing the next line into the `Haskell` code:

```
{-# LANGUAGE TemplateHaskell, QuasiQuotes #-}
```

3.3 Step 3. Applying the *Sobel Operator*

Following the function `applyEdgeDetection` definition, the *Gaussian filter* image-result is passed as parameter to the function `applySobel`. This is responsible for the application of the *Sobel operator*. The code is listed below.

```
type DelayedImageC1 = Array D DIM2 Float

computeMagnitude :: Float -> Float -> Float
{-# INLINE computeMagnitude #-}
computeMagnitude dx dy = sqrt(dx * dx + dy * dy)

computeDyImage :: ImageC1 -> IO ImageC1
{-# INLINE computeDyImage #-}
computeDyImage image = computeP (
                         forStencil2 (BoundConst 0) image
                            [stencil2| -1 0 1
                                       -2 0 2
                                       -1 0 1 |])

computeDxImage :: ImageC1 -> IO ImageC1
{-# INLINE computeDxImage #-}
computeDxImage image = computeP (
              forStencil2 (BoundConst 0) image
                     [stencil2| -1 -2 -1
                                 0  0  0
                                 1  2  1 |])

computeMagnitudeImage :: ImageC1->ImageC1->Float->DelayedImageC1
{-# INLINE computeMagnitudeImage #-}
computeMagnitudeImage dxImage dyImage threshold =
     R.map (\ magnitude -> if magnitude < threshold then 0 :: Float
                     else if magnitude > 1 :: Float then 1 :: Float
                           else magnitude)
            (R.zipWith computeMagnitude dxImage dyImage)

applySobel :: Float -> ImageC1 -> IO DelayedImageC1
applySobel threshold image = do
    dxImage <- computeDxImage image
    dyImage <- computeDyImage image
    return (computeMagnitudeImage dxImage dyImage threshold)
```

In this code we note that the function `applySobel` returned type is `IODelayedImageC1` because the functions `R.map` and `R.ziptWith` return type `D` vectors as `Delay` arrays. These arrays are considered similar with functions because they determine the value of the elements only by request.

In the previous examples, each time the `function R.map` is applied, the function `computeP` is called, too. Now, it is not necessary to do the same because the function converting the image from `DelayedImageC1` in `ImageC3` will be called instead. For this, it needs just the value of a single element at a certain moment, contrasting with the previous situations.

Finally, the resulting image will be saved on the disk following the path specified by user. Running the compiled program is done using the command:

```
EdgeDetector +RTS -N (-s)
```

4 Final results. Comparative aspects

With respect to this paper target, namely the evaluation of the performance of a `Haskell` parallel implementation, we have tested our developed application on a 3008×2000 image using a computer with two cores. Technical specifications for the computer system used for tests are: Pentium® Dual-Core CPU T4300 @ 2.10 GHz 2.10 GHz. The corresponding execution times are shown in Table 1.

In order to have reliable conclusions, we decided to develop the same applications making use of one of the most popular programming languages, namely `C++`. The `C++` project is developed in `VS2012`, `MFC` and `OpenCV 2.4` for reading and printing the images. Beyond the serial implementation, for consistent results, we have also implemented a `C++` solution based on the parallel programming support of this language. Particularly, we used two different options for parallel excution, namely: `PPL` (*Parallel Patterns Library*) and `OpenMP` (*Open MultiProcessing*) [11]. It is our aim to publish this `C++` project details and its related results in [10]. The final results are also presented here in Table 1 in order to compare them to the `Haskell` implementation execution times.

Totally, a number of 11 tests were made for each presented case and the arithmetic average of the returned values has been set in the table. The following masks have been applied: $G_{3\times3}$ - for *Gaussian filter* dimensions 3×3, $G_{5\times5}$ - for *Gaussian filter* dimensions 5×5 and $G_{7\times7}$ - for *Gaussian filter* dimensions 7×7. The `Repa` library has these mask sizes as default dimensions, that is why only the sizes shown in Table 1 are supported for the moment. In the table, the best time is marked with bold and the worst value is marked with both italic and underline.

Table 1: **Execution time**

Algorithm	$G_{3\times3}$	$G_{5\times5}$	$G_{7\times7}$
Haskell	**226 ms**	457 ms	*4999 ms*
C++ Serial	*555 ms*	*732 ms*	995 ms
C++ PPL	438 ms	513 ms	626 ms
C++ OpenMP	295 ms	**385 ms**	**533** ms

An important conclusion from this table is that the algorithm written in `Haskell` is the fastest for $G_{3\times3}$, followed closely by the `OpenMP` solution. In this case, the `PPL` implementation is quite slow.

For $G_{5\times5}$ the hierarchy changes. This time, `OpenMP` is the fastest. This is followed by the code written in `Haskell`. Here, as well, `PPL` is much slower. The delay is probably given by the time necessary to manage the threads execution because the results of an execution time profile show that a function in the `PPL` library is slower. In practice, this dimension of the *Gaussian filter* is most often used, as it is also specified in the documentation associated to the `Repa` library. Consequently, this could be the most significant result.

In the latter case, the fastest is still `OpenMP` followed by `PPL` and then by the serial algorithm. Surprisingly, the code written in `Haskell` is too slow for this dimension. It is true that applying the $G_{5\times5}$ twice yields the same result as applying the $G_{7\times7}$ once. Nonetheless, the difference is too big.

5 Conclusions

In this paper, the specific case of a `Haskell` parallel implementation for an image processing context is considered. This implementation is supported by the `Repa` library through its free set of combinators for building parallel array computations. From the practical point of view, the edge detection problem has been chosen. Our targets were both to discuss the parallel implementation in `Haskell` for this problem solution and to compare the execution time values for this solution to the results returned by the similar implementations in `C++`.

Here, it yields that we have not an all-purpose answer. The compared execution time values depend both on the physic and the logic parameters of the applied solutions, such as: the processor architecture, the default constraints established by the software libraries and the specific package which is effectively used for implementations.

The comparison of `Haskell` and `C++` support for different computing fields is a subject of interest for modern research areas, both theoretical and practical. For example, in [8], [3] many relevant aspects are revealed by comparing the facilities offered by many programming languages, including `Haskell` and `C++`, for generic programming. By implementing a substantial example in each of these languages, the authors illustrate in [8] how the basic roles of generic programming can be represented in each language. Besides, for assessing how well a language supports generic programming, in [5] the authors propose a taxonomy that captures commonalities and differences between specific programming issues in `Haskell` and `C++`.

As a perspective, we would like to extend our work on `Haskell` to some pattern mining algorithms. We intend to use specific libraries, like `HLearn` [2], [14], the recently published `Haskell` library for machine learning. The `HLearn` distinguishing feature is that it exploits the algebraic properties of learning models and its goal is to make machine learning techniques easily usable for non specialists. Continuting our research results from [6], an interesting perspecive could be to find a `Haskell` alternative solution for practical problems basically solved with `LAD` (*Logical Analysis of Data*) using `HLearn`.

References

[1] Marlow, S., *Parallel and Concurrent Programming in Haskell*, O'Reilly Media, Inc, 2013.

[2] Izbicki, M., *HLearn: A Machine Learning Library for Haskell*, Proceedings of The Fourteenth Symposium on Trends in Functional Programming, Brigham Young University, Utah, May 14-16, 2013.

[3] Oliveira B.C.D.S., Schrijvers, T., Choi W., Lee, W., Yi, K., (2012), *The implicit calculus: a new foundation for generic programming*, Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, SESSION: Foundations, Volume 47 Issue 6, June 2012, 35-44, 2012.

[4] Lippmeier, B., and Keller, G., *Efficient Parallel Stencil Convolution in Haskell*, Haskell '11, ICFP 2011, Proc. of the 4th ACM symposium on Haskell, SESSION: Parallelism, Tokyo, Japan, 22nd September, 2011, 59-70, 2011.

[5] Bernardy, J.-P., Jansson, P., Zalewski, M., and Schupp, S., *Generic programming with C++ concepts and Haskell type classes-a comparison*, Journal of Functional Programming, **20**, (2010), 271-302.

[6] Băicoianu, A., Dumitrescu, S., *Data mining meets economic Analysis: Opportunities and Challenges*, Bulletin of the *Transilvania* University of Braşov, Series V: Economic Sciences, Vol. 3, **52**, (2010), 185-192.

[7] Keller, G., Chakravarty, M. M., Leshchinskiy, R., Peyton Jones S., and Lippmeier, B., *Regular, Shape-polymorphic, Parallel Arrays in Haskell*, Proc. of the 15th ACM SIGPLAN International Conference on Functional Programming, Baltimore, MD, USA, September 27 - 29, 2010, 261272, 2010.

[8] Garcia, R., Jarvi, J., Lumsdaine, A., Siek J., and Willcock J., *An extended comparative study of language support for generic programming*, Journal of Functional Programming, **17**, (2007), Issue 02, 145-205.

[9] Acharya, T., Ray, A.K., *Image Processing. Principles and Applications*, Wiley Interscience, New Jersey, August 2005.

[10] Băicoianu, A., Pândaru, R., Vasilescu, A., *Upon the performance of a C++ parallel implementation*, preprint.

[11] ***, *MSDN Library*, on-line http://msdn.microsoft.com/en-us/library

[12] ***, *Haskell 98 Language and Libraries*, The Revised Report, December 2002, also on-line at http://www.haskell.org/onlinereport

[13] ***, *The Glorious Glasgow Haskell Compilation System*, User's Guide, Version 7.6.3, also on-line at http://www.haskell.org/ghc/docs/7.6.3/html/users_guide

[14] ***, *The Haskell available packages*, on-line http://hackage.haskell.org