Using Erlang Skeletons to Parallelise Realistic Medium-Scale Parallel Programs

(Early Draft!)

Vladimir Janjic   Christopher Brown   Kevin Hammond

University of St Andrews, Scotland, UK.

{vj,cmb21,kh}@st-andrews.ac.uk

Abstract

This paper shows how the Erlang skeleton library, *Skel*, can be used to parallelise the Discrete Haar Wavelet Transform application. The Discrete Haar Wavelet Transform is a very important wavelet transformation, which is heavily used in image and signal processing. Using the Skeleton version of the application, we were able to achieve speedups of 16.63 on a 24-core shared memory machine. This demonstrates that, with relatively little effort, the *Skel* library can be used for parallelisation of Erlang applications, obtaining very good speedups.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords Erlang, Parallelism, Skeletons

1. Introduction

The single-core processor, which has dominated for more than half a century is now obsolete. Machines with dual-, quad- and even hexa-core CPUs are already common place in desktop machines and CPUs with 50 cores as standard have already been announced [1]. There has been a *seismic* shift between sequential and parallel hardware, but programming models have been very slow to keep pace. Indeed, many programmers still use outdated sequential models for programming parallel systems, where parallel concepts have effectively been *bolted-on* to the language, rather than high-level parallel constructs being *first-class*. What is needed is an effective solution to help programmers *think parallel*. In the context of parallel programming, parallel design patterns represent a natural language description of a recurring problem and of the associated solution techniques that the parallel programmer may use to solve that problem.

Porting existing sequential applications to large-scale shared-memory parallel systems usually comprises of *identifying* where in the application the potential parallelism lies and then, subsequently, *implementing* low-level parallel code that exploits this parallelism in a useful manner. The implementation is usually very tedious and error-prone, since the programmer usually needs to explicitly handle thread creation, communication and synchronisation. *Algorithmic Skeleton Libraries* [7] attempt to abstract away from these tedious details, by providing a set of high-level *skeleton functions* that capture common parallelism patterns. The user then only needs to provide the sequential code for the skeleton(s) that he choses, and the low-level parallel details (such as thread creation, communication and synchronisation) are abstracted away.

In this paper we will introduce a number of new, "real world" use cases, demonstrating the effectiveness of using Erlang as a parallel programming language. Our use cases build on top of a recent new skeleton framework for Erlang, Skel, which provides a set of classical, well-founded parallel implementations of the most well known, and useful skeletons. We show, for each use case, its performance results on a 24-core shared memory machine.

In the full paper, we promise to show

- A number of realistic use-cases implemented using the Skeleton library, Skel in Erlang for parallel shared memory systems. The use cases will include Ant Colony Optimization; Dialyzer, Image Processing and Haar Transformation.
- An evaluation of the use-cases on a number of different shared-memory many-core platforms.
- The identification of new parallel skeletons, by evaluation of the use cases.

2. Background

2.1 Erlang

Erlang is a strict, impure, functional programming language with support for *first-class* concurrency. This concurrency model allows the programmer to be explicit about processes and communication, but implicit about placement and synchronisation. Erlang supports a *lightweight* threading model, where processes model small units of computation (tasks) that are executed on a capability. The scheduling of processes is handled automatically by the Erlang Virtual Machine, which also provides basic load balancing mechanisms. Erlang typically has three primitives for handling concurrency:

- spawn(), allowing new functions to execute in a lightweight Erlang process;
- !, allow messages to be explicitly sent from one Erlang process to another; and,
- receive, to allow messages to be received in another process queue.

[1] Intel's Many Integrated Core Family

CONF 'yy, Month d–d, 20yy, City, ST, Country.
Copyright © 20yy ACM 978-1-nnnn-nnnn-n/yy/mm...$15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Furthermore, Erlang also supports fault tolerance, by allowing groups of processes to be *supervised*, and new instances of processes can be spawned in the case failure. Although Erlang supports concurrency, there has been little research into how Erlang can be used to effectively support *deterministic* parallelism.

2.2 Skeletons

An *algorithmic skeleton* [7] is an abstract computational entity that models some common pattern of parallelism (such as the parallel execution of the sequence of computations over the set of inputs, where the output of one computation is the input to the next one). A skeleton is typically implemented as a high-level function that takes care of the parallel aspects of a computation (e.g., the creation of parallel threads, communication and synchronisation between these threads, load balancing etc.), and where the programmer supplies sequential problem-specific code and any necessary skeletal parameters.

Skel [5], is a Domain Specific Language implemented in Erlang for expressing parallelism using algorithmic skeletons. It currently provides a small number of classical skeletons that are considered the most useful, including Map, Farm, Pipeline and Seq. In this paper, we consider two skeletons:

- `seq` is a trivial wrapper skeleton that implements the sequential evaluation of a function, $f :: a- > b$, applied to a sequence of inputs, $x_1, x_2, \ldots, x_n$.
- `pipe` models a parallel pipeline applying the functions $f_1, f_2, \ldots, f_m$ in turn to a sequence of independent inputs, $x_1, x_2, \ldots, x_n$, where the output of f_i is the input to f_{i+1}. Parallelism arises from the fact that, for example, $f_i(x_k)$ can be executed in parallel with $f_{i+1}(f_i(x_{k-1}))$. Here, each f_i has type $a- > b$.
- A `farm` skeleton models the application of a single function, $f :: a- > b$, to a sequence of independent inputs, $x_1, x_2, x_3, \ldots, x_n$. Each of the $f(x_1), f(x_2), f(x_3), \ldots, f(x_n)$ can be executed in parallel.
- A `map` skeleton is a variant of a farm where each independent input, x_i, can be $x_1, x_2, x_3, \ldots, x_n$, is partitioned ($p :: a- > [b]$) into a number of sub-parts that can be worked upon in parallel, a worker function, $f :: [b]- > [c]$, is then applied to each element of the sublist in parallel, finally the result is combined ($c :: [c]- > d$) into a single result for each input.

3. Ant Colony

Ant Colony Optimisation (ACO) is a heuristic for solving NP-complete optimisation problems, inspired by the behaviour of ants living in real ant colonies. An ACO algorithm consists of a number of iterations. In one iteration, each ant independently computes a solution to the problem, with the solution being partially guided by a *pheromone trail* produced by ants. To compute one component of a solution, an ant (with the designated probability q) follows the pheromone trail for that component or (with the probability $1 - q$) it performs a biased random selection of the component. In this way, different ants generally produce different (but similar) solutions. After the iteration is finished, and all ants have computed solutions, the best solution is chosen and the pheromone trail is updated according to that solution. After that, the next iteration, where ants compute new solutions based on the new feromone trail, is started.

An example ACO algorithm that we consider in this deliverable is computing a solution to a Single Machine Total Weighted Tardiness Problem (SMTWTP). In SMTWTP, we are given n jobs, whereas each job i is characterised by its processing time, p_i, deadline, d_i, and weight, w_i. The goal is to schedule execution of jobs in a way that achieves minimal total weighted tardiness. The tardiness of a job ,i, in a schedule is defined by $T_I = \max\{0, C_i - d_i\}$, where C_i is the completion time of the job i in that schedule. The total tardiness of the schedule is defined as $\sum w_i T_i$.

In the ACO solution to the SMTWTP problem, in each iteration each ant independently computes a schedule. The pheromone trail that guides the computation of schedules is defined by a matrix π, where $\pi[i, j]$ is the preference of assigning job j to the i-th place in the schedule. Therefore, in each step of the solution computation, an ant will either pick the job with the highest preference for that position, or will choose a biased random selection (again based on the pheromone trail). Once the iteration is finished and all ants have computed their schedules, the schedule that obtains the minimal total weighted tardiness is selected, and the pheromone trail is modified to increase chances of selecting job in the same order as in the currently found best solution.

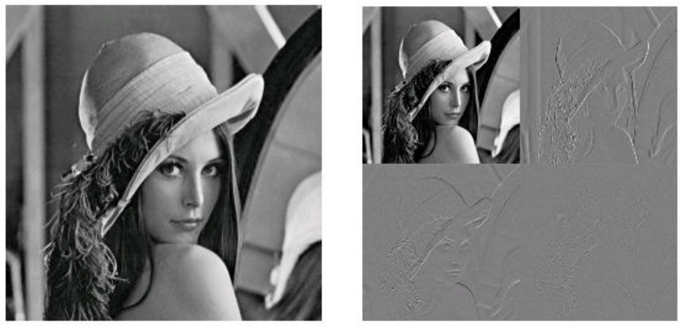
(a) Before (b) After

Figure 1. An image before and after the Discrete Haar Transformation

4. Dialyzer

5. Merging and Filtering

6. The Discrete Haar Wavelet Transform

In mathematics, the *Haar wavelet* [11] is a sequence of “square-shaped” functions that can be used to approximate any square-integrable real function. An important example of the use of Haar wavelets in Computer Science is the *Discrete Haar Transform*, which is heavily used in image and signal processing. The Discrete Haar Transform consists of applying the operation

$$y = TxT^T, \tag{1}$$

for an input vector x and a fixed Haar matrix $T = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$, producing the output vector y. A 1-dimensional Discrete Haar Transform, where x is a pair of real numbers, is used in signal processing (e.g. sound compression). A 2-dimensional Discrete Haar transform, on the other hand, where x is 2x2 matrix, is used for image compression, as for each 2x2 matrix of pixels x, it gives a 2x2 matrix of pixels y as a result, where most of the energy of x is contained in the top left pixel of y. Applying the Discrete Haar Transform to an image (signal) consists of splitting the image (signal) into 2x2 subimages (pairs of elements) and applying the Haar Transform to each subimage (pair). See Figure 1 for the example of an image before and after the Discrete Haar Transform is applied.

We consider two use cases of applying the Discrete Haar Transform:

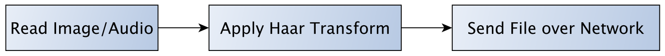

Figure 2. Basic Sequential Haar Transform

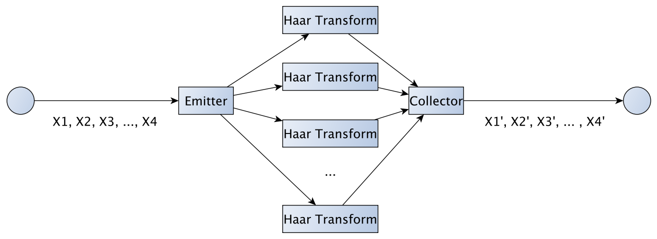

Figure 3. Haar Transform Using a Skeleton Task Farm

1. *audio compression*, where a 1D Discrete Haar Transform is applied to a stream of audio files; and,
2. *video compression*, where a 2D Discrete Haar Transform is applied to a stream of images captured by a camera.

We can observe two sources of parallelism in these use cases. At the top level, we have *outer-level* parallellism, where the same operation (Discrete Haar Transform) is applied to a set of independent inputs (a stream of audio files or images). This is relatively coarse-grained data-parallelism, which can be implemented using a task farm. In the application of the Discrete Haar Transform to one audio file or image, we can observe the *inner-level* parallelism, where the same operation 1 is applied to a set of independent subvectors or submatrices of an original audio file or image. This represents much finer-grained data parallelism.

In this paper, we exploit only the outer-level parallelism in the two use cases, using a *Farm* Erlang skeleton. We leave exploitation of inner-level parallelism as future work, where we plan to use an *OpenCL* kernel for dealing with fine-grained data-parallelism in applying the Discrete Haar Transform to one audio file/image, and to use Erlang/*OpenCL* bindings to execute this kernel within the Erlang Virtual Machine. This *OpenCL* binding will be wrapped inside a skeleton structure where both the *OpenCL* version and the Erlang version can be operated in parallel, allowing us to study the effectiveness of Heterogeneity in Erlang.

6.1 Implementing the Haar in Skel

In this section we illustrate the process of porting the sequential Haar use-case to the Erlang parallel skeleton library, *Skel*. The basic program comprises a three-stage function composition (see Figure 2). In the first stage, a stream of files is read, where each file comprises a video or an audio file. These audio files/images are then passed to the second stage, where the Discrete Haar Transform is applied (1D for audo files and 2D for images). Finally, in the third stage, the transformed audio files/images are sent across a network for further processing (if necessary). In order to parallelise this algorithm, we decided to introduce a task farm for the second stage (see Figure 3). In order to do this, we created a Skeleton

Algorithm 1 Basic Sequential Algorithm for 1D Haar Transform at the *Outer* Level

```
sequential1d(Vectors) ->
    [haar_1d_wrapper(V) || V <- Vectors].

haar_1d_wrapper({R, A, I, Lim}) ->
     seq_haar_1d_a(R, A, I, Lim).
```

Algorithm 2 1D Haar Transform Using a Skel Task Farm

```
theSkel1D(Vectors) ->
   skel:run([{farm, [{seq,
                      ?MODULE:haar_1d_wrapper/1}],
                      24}],
                      Vectors),
   receive
        {sink_results, Results} -> Results
   end.
```

call to a Skel task farm, where the number of farm workers is controlled by the user via an input argument. A task corresponds to applying sequentially the Discrete Haar Transform to one audio file/image. The tasks $X_1, X_2, \ldots, X_n$ are distributed to workers using a farm *emitter*, in an on-demand fashion (where tasks are sent to idle workers). The task results $X_1', X_2', \ldots, X_n'$ are sent to the farm *collector*. Both the emitter and collector are hidden from the user, and are provided by the basic library framework.

6.2 1D Haar Transform

The porting of the 1D Haar Transform proceeded in two stages, where the original algorithm is shown in Listing 1. Here the sequential 1D Haar Transform, sequential1d, is defined as mapping a function, haar_1d_wrapper, over a list of Vectors.

Stage 1: Introducing a Task Farm Introducing parallelism into the program was done by first identifying the sub-expression in the program that generated the output list, and where each operation on the list could be computed in parallel. In the 1D Haar Transform example, the operation sequential1d is converted into a task farm, where each worker computes the 1D Haar Transform for an input vector. The result of the task farm is a list of transformed vectors, as illustrated in Listing 2.

Here, the program can be broken down into a number of key components:

- skel:run denotes a call to the top-level run function in the skel library, where the parameter to run is a (nested) Skeleton;
- farm denotes a farm skeleton;
- seq denotes the workers of the farm skeleton are the sequential function, 1d_haar_wrapper;
- 24 denotes the number of farm workers; and, finally,
- Vectors is the input list of tasks.
- sink_results is an Erlang atom that is used to determine that we have a result from the skeleton. The Erlang receive block waits for a message to be returned from the skeleton, matching sink_results, and binding the result to the variable, Results.

Stage 2: Chunking While using a task farm for the 1D Haar Transform creates a reasonable amount of parallelism, the parallelism is too fine-grained and the program does not scale as we would typically expect. This is a common problem in the early stages of writing parallel programs. To combat this, we introduce

Algorithm 3 1D Haar Transform Using a Skel Task Farm with Partitioning and Combining

```
theSkelChunked(C, Vectors, Len) ->
    skel:run([{farm, [{seq, fun(V) ->
             (lists:map(fun(X) ->
                    haar_1d_wrapper(X) end,V)) end}],
                    24}],
                    partition(Vectors, C, Len)),
    receive
      {sink_results, Results} -> combine(Results)
    end.
```

Algorithm 4 Partition and Combine Functions in Erlang

```
combine([])->[];
combine([X|Xs]) -> lists:append(X, combine(Xs)).

partition([], ChunkSize, Len) -> [];
partition(List, ChunkSize, Len) ->
     case (length(List) < ChunkSize) of
       true -> [List];
       false -> Chunk = lists:sublist(List,
                                     ChunkSize),
                    NewList = lists:sublist(List,
                                     ChunkSize+1,
                                     Len),
                    [Chunk | partition(NewList,
                                  ChunkSize,
                                  Len)]
     end.
```

Algorithm 5 Partition and Combine Functions in Erlang

```
sequential2D(Images) -> [ haar_2d_wrapper(I)
                          || I <- Images].
```

chunking to the task farm, allowing us to group together a number of small tasks into one larger parallel task, where each parallel thread operates over a sub-list rather than just one element. We want each worker to be busy, so we chunk by groups of 4 elements, $(2048/4 = 512$ tasks for each worker). By chunking in this way, we also decrease the communication costs, and reduce parallel overheads. Chunking can generally be achieved in a variety of different ways. In our example, we modify the task farm, manually refactoring it to a pipeline with a partition and combine stage, as illustrated in Listing 3.

We also have to introduce two new functions, partition and combine, as illustrated in Listing 4. Here, combine simply takes a list, where the head of the list, X, is also a list, and appends X to the combined tail of the list, Xs. Partitioning a list is implemented by the partition function, where a new list is created that is ChunkSize elements in length. This sublist is then appended to the remaining sublists of the input list, List. The partitioning terminates when no new sublists can be created.

6.3 2D Haar Transform

The porting process for the 2D Haar Transform proceeded in a similar way to the 1D Haar Transform, where the sequential algorithm is shown in Listing 5.

Here, a function, haar_2d_wrapper is applied to each element, I, of the list of input images, Images. Porting this code to use a skel

Algorithm 6 Partition and Combine Functions in Erlang

```
theSkel2d(Vectors) ->
    skel:run([{farm, [{seq,
                         fun ?MODULE:2d_wrapper/1}],
                         24}], Vectors),
    receive
         {sink_results, Results} -> Results
    end.
```

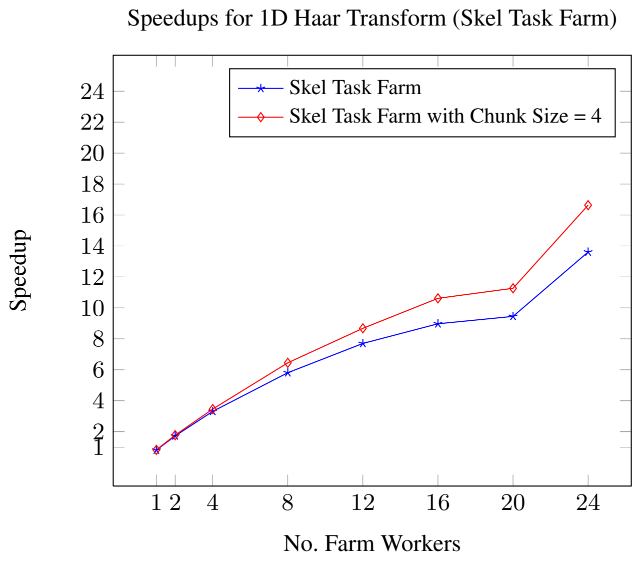

Figure 4. Speedup figures for a 1D Haar Transform, for 2048 audio files, with a sample size of 4400.

task farm comprised of rewriting the above code into a call to the skel library, introducing a farm skeleton, where the work function is 2d_wrapper, as shown in Listing 6.

Here we introduce a call to skel:run, which is the top-level skeleton call, parameterised by a nested skeleton. In our example, we use the farm skeleton, with 24 workers, and each worker is a sequential function, 2d_wrapper. In this example, it is not necessary to employ chunking, as the tasks are already large enough to give sufficiently large computation, without saturating the system with an abundance of parallel tasks.

7. Evaluation Results

In this section we evaluate our 1D and 2D Haar Transformations, where all measurements have been made on an 800 Mhz 24 core, dual AMD Opteron 6176 architecture, running Centos Linux 2.6.18-274.el5. and Erlang 5.9.1 R15B01, averaging over 10 runs. We report absolute speedups against the original sequential versions.

7.1 1D Haar Transform

For the 1D Discrete Haar Transform, we executed the application over 2048 audio samples, each with a sample size of 4400. This translates to 2048 vectors, each with 4400 elements, where a single execution of a 1D Discrete Haar Transform operates over a single vector. Figure 4 shows the speedup results for the parallel version of the 1D Discrete Haar Transform using 1 – 24 farm workers. In the figure, the blue line corresponds to the speedups of the simple task

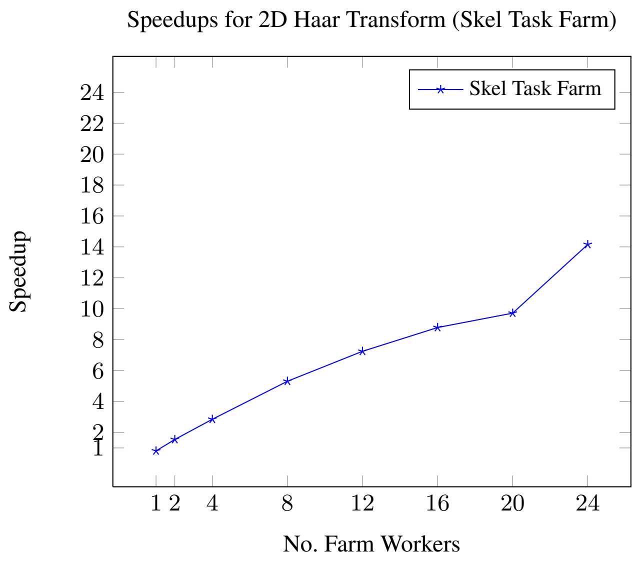

Figure 5. Speedup figures for a 2D Haar Transform, for 24 images, 1024*1024.

farm variant of the application, where a task corresponds to processing one vector. The red line shows the chunked version (where the input list of vectors is partitioned into groups of 4 vectors). The chunked version still uses a task farm, but here one tasks corresponds to processing 4 vectors. The simple task farm variant (blue line) shows a maximum speedup of 13.6, where the version with chunking shows an improvable speedup of 16.63. Although more investigation is needed at this stage, we speculate that in the simple task farm variant, there is an abundance of parallelism, where the system is saturated with many fine-grained tasks. In the chunking version, the number of tasks is reduced, but the computation size is increased, therefore reducing the communication overheads.

7.2 2D Haar Transform

Figure 5 shows speedup results for the 2D Haar Transform, over 24 images, each $1024*1024$ in size. Here, we only consider the simple task farm without chunking. As it can be seen, the application starts to scale reasonably well, tailing off with around 12 workers with a speedup of 7.2. This is due to a fact that there are 24 tasks, so with 12 workers, each worker gets exactly 2 tasks. Increasing the number of workers to 13 (and up to 23) results in imbalance in the number of tasks allocated to workers. Therefore, some workers may be idle for considerable time, waiting for workers that got more tasks to finish. This further results in only small improvements in speedups when there are between 13 and 23 workers. Balance is again restored when there are 24 workers, hence the much better speedup of 14.15.

8. Related Work

The Skel framework was introduced in [5], together with a methodology for parallelising Erlang programs using refactoring tools and cost-models. In this paper we attempted to follow the methodology, replacing the refactoring tool-support with a manual refactoring process instead. Since the nineties, the "skeletons" research community has been working on high-level languages and methods for parallel programming [3, 4, 6–9]. Skeleton programming requires the programmer to write a program using well-defined abstractions (called skeletons) derived from higher-order functions that can be parameterized to execute problem-specific code. Skeletons do not expose to the programmer the complexity of concurrent code, for example synchronization, mutual exclusion and communication. They instead specify abstractly common patterns of parallelism – typically in the form of parametric orchestration patterns – which can be used as program building blocks, and can be composed or nested like constructs of a programming language. A typical skeleton set includes the pipeline, the task farm, map and reduction. There has been a few previous attempts at parallelising Erlang applications, such as parallelising Dialyzer [1], and a suite of Erlang benchmarks [2]. However, none of the attempts exploit structured parallelism in the form of algorithmic skeletons, as outlined in this paper. Parallelism has been exploited in other functional languages, such as Haskell, using a strategies approach for implicit parallelism in GpH [12], and an explicit structured parallelism approach, using algorithmic skeletons, for Eden [10].

9. Conclusions and Future Work

We have presented evaluation results for a 1D and 2D Discrete Haar Wavelet Transform. For the 1D Discrete Haar Transform, we presented two variants, both using a task farm skeleton from the Erlang Skel library. The first variant was a basic task farm skeleton with 24 workers, and the second variant was a task farm skeleton with partitioning. Our evaluation showed that the basic task farm variant achieves a speedup of 13.6 on a 24-core shared-memory machine, where the partitioning version gives an improved speedup of 16.6. The partitioning version gives a 21% increase in performance over the basic task farm variant. For the 2D Haar Transform, we evaluated the application using a simple task farm skeleton from the Erlang Skel library, demonstrating a speedup of 14.15 of 24 cores. Clearly this shows reasonable and scalable speedups of the Haar Transform use case. For future work, we intend on performing more experiments on a range of platforms, and also evaluating an OpenCL variant, which would also allow us to take into account GPU architectures.

Acknowledgments

This work has been supported by EU Framework 7 grants IST-288570 "ParaPhrase: Parallel Patterns for Adaptive Heterogeneous Multicore Systems" (`http://www.paraphrase-ict.eu`), and IST-248828 "Advance: Asynchronous and Dynamic Virtualisation through performance ANalysis to support Concurrency Engineering (ADVANCE)" (`http://www.project-advance.eu`).

References

[1] S. Aronis and K. Sagonas. On using erlang for parallelization : Experience from parallelizing dialyzer.

[2] S. Aronis, N. Papaspyrou, K. Roukounaki, K. Sagonas, Y. Tsiouris, and I. E. Venetis. A scalability benchmark suite for erlang/otp. In *Proceedings of the eleventh ACM SIGPLAN workshop on Erlang workshop*, Erlang '12, pages 33–42, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1575-3. . URL `http://doi.acm.org/10.1145/2364489.2364495`.

[3] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^{3}L: A Structured High level programming language and its structured support. *Concurrency Practice and Experience*, 7(3):225–255, May 1995.

[4] G. H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proc. of the 5th International Symposium on High Performance Distributed Computing (HPDC'96)*, pages 243–252. IEEE Computer Society Press, 1996.

[5] C. Brown, M. Danelutto, P. Kilpatrick, K. Hammond, and A. Elliott. Cost directed refactoring for parallel erlang programs. *International Journal of Parallel Programming. To Appear*, 2013.

[6] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computations*. Research Monographs in Par. and Distrib. Computing. Pitman, 1989.

[7] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30(3):389–406, Mar. 2004. ISSN 0167-8191. . URL `http://dx.doi.org/10.1016/j.parco.2003.12.002`.

[8] J. Darlington, Y. Guo, Y. Jing, and H. W. To. Skeletons for structured parallel composition. In *Proc. of the 15th Symposium on Principles and Practice of Parallel Programming*, 1995.

[9] M. Hamdan, P. King, and G. Michaelson. A scheme for nesting algorithmic skeletons. In K. Hammond, T. Davie, and C. Clack, editors, *Proc. of the 10th International Workshop on the Implementation of Functional Languages (IFL'98)*, pages 195–211. Department of Computer Science, University College London, 1998.

[10] R. Loogen, Y. Ortega-mallén, and R. Peña marí. Parallel functional programming in eden. *J. Funct. Program.*, 15(3):431–475, May 2005. ISSN 0956-7968. . URL `http://dx.doi.org/10.1017/S0956796805005526`.

[11] P. Porwik and A. Lisowska. The Haar-Wavelet Transform in Digital Image Processing: Its Status and Achievements. *Machine Graphics and Vision*, 13:79–98, 2004.

[12] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60, Jan. 1998. ISSN 0956-7968. . URL `http://dx.doi.org/10.1017/S0956796897002967`.