Using algorithmic skeletons in EcmaScript to parallelize computation in browsers through Web Workers

Damian Schenkelman and Matias Servetto

School of Engineering. University of Buenos Aires.
Paseo Colon 850, Buenos Aires, Argentina
{dschenkelman,mservetto}@fi.uba.ar
http://www.fi.uba.ar/

Abstract. With the growing tendency of browsers being used for CPU intensive applications (such as 3D games) it is important to take advantage of multiple cores to reduce user perceived latency. This paper presents a JavaScript library based on algorithmic skeletons that allows users to take advantage of parallel processing in browsers while maintaining a familiar coding style.

Keywords: browsers, ecmascript, parallelization, algorithmic skeletons

1 Introduction

In 1965 Gordon Moore proposed the popularly known "Moore's law" [1], which predicts that the number of transitors in integrated circuits will double every two years. As a consequence, programs that use a single processor/core will automatically become faster without the need to modify them with that purpose in mind.

In the year 2010 it was predicted that this tendency would be slowly reaching its end, mostly due to reasons related to heat dissipation. This is the cause why newer computer have a greater number of processors instead of processors with more computing power.

To be able to maximize the performance of these computers, it is paramount that systems professionals can create programs that process data in parallel, through different mechanisms, such as threads or processes executing simultaneously in different processors. In the case of application executed on a single device (e.g. browsers and web pages, applications for mobile devices, desktop apps, server applications) a lot of languages and platforms provide a simple way for programmers to abstract the complexity and coordination required for this type of processing. For example, the .NET platform has the Task Parallel Library (TPL) and Parallel LINQ (PLINQ) [2] libraries that use algorithmic skeletons modelled with high order functions so developers can perform complex operations through a simple API.

On a different note, one of the tendencies that is beggining to take off is the use of EcmaScript [3] (also known as Javascript) to create applications that were previously only considered viable in a native environment, such as video games. This has been possible thanks to components like asm.js [4] and WebGL [5] together with the constant evolution of browsers and JavaScript runtime engines. In this context, one of the plans of the commitee that develops the language is to provide an API to simplify the processing of data in parallel for version 7 (ES7) of the language, with the goal of keeping up with native applications. Initiatives such as ParallelJS [7] and River trail [6] or the possibility of taking advantage of SIMD instructions [8] (single instruction multiple data) are some of the options to implement this.

The goal of this paper is to provide an alternative for parallel code execution in EcmaScript, through a library that exposes high order functions to model algorithmic skeletons such as map, filter and reduce. This library will take advantage of Web Workers [9] (the mechanism proposed by browsers for parallelism at the time of writing) to allow the simultaneous execution of code in multiple processors.

2 Background

All modern versions of major browsers (Internet Explorer, Mozilla Firefox, Google Chrome, Safari and Opera) allow the execution of JavaScript code in a single threaded event loop runtime [10]. User actions add elements to the event loop queue and these are processesed sequentially. All process input/output is asynchronous to prevent the UI thread from blocking, thus keeping it responsive. While such a runtime is a good fit for most web applications, JavaScript and the browser are now also being used for some CPU intensive tasks, such as games, which commonly require physics calculations, image manipulation (in 2D) and graphics generation (in 3D). Despite the large improvement in JavaScript performance obtained through optimizing compilers such as Google Chrome's V8 [11] and Mozilla Firefox's SpiderMonkey [12], it is important to be able to take advantage of multiple cores in modern devices to achieve even greater performance.
In 2010, Web Workers were made part of the web standard. Web Workers allow the creation of "thread like" constructs in a browser environment, but they don't allow on shared memory and instead communicate via message passing. The message passing overhead is very big for common objects and Web Workers have a considerable startup time, so they were commonly considered for long running tasks and operating on generally the same set of data during a single execution, making them unfit for a thread pool model [13].
However, a recent proposal for version 7 of the EcmaScript standard changes this, by introducing the notion of shared memory through `SharedArrayBuffer`s. In essence, the proposal allows the same memory to be shared across multiple Web Workers, and also aims to provide the necessary atomic/lock constructs to deal with shared memory. In the particular case of embarrassingly parallel compu-

tations (such as map, filter and reduce operations on an array), it is simple to take advantage of the shared memory speed benefits without incurring in any overhead due to locks.

3 Serialization and Transference

Let's consider an array of `N` elements on which a particular transformation is to be performed through the `map` function. If the transformation is executed using a single thread (no parallelism) and the average time to process each element is `t` then the total time (T_{ser}) for the operation can be approximated as:

$$T_{ser} = \sum_{i=0}^{N} t = Nt \; . \tag{1}$$

When trying to parallelize this operation using K threads the ideal goal is to reach a total time (T_{par}) that is:

$$T_{par} = \frac{T_{ser}}{K} \; . \tag{2}$$

Nevertheless, there are additional time consuming tasks other than the main computation that need to be considered when performing the operation in parallel in programs where not all memory is shared. These are:

- Serializing/deserializing the elements to transfer.
- Transferring the elements back and forth between the UI thread and the workers.
- Serializing/deserializing the function to transfer.
- Transferring to each worker the functions for the transformation.

If we drill down into the different parts:

- T_{ft} as the function transfer time
- T_{et} as the elements transfer time
- T_{fs} as the function serialization/deserialization time
- T_{es} as the elements serialization/deserialization time

It is clear that:

$$T_{sync} = T_{ft} + T_{et} + T_{fs} + T_{es} \; . \tag{3}$$

$$T_{par} \approx \frac{T_{ser}}{K} + T_{sync} \; . \tag{4}$$

Based on Eq. 3 and Eq. 4 it can be deduced that the more T_{sync} can be reduced, the closer to the ideal scenario the computation will be.

In our case we are trying to transfer objects between a browser's JavaScript UI thread and Web Workers so we are constrained by the means of that environment. The Worker interface is the following [9]:

Web Worker interface

```
[Constructor(DOMString scriptURL)]
interface Worker : EventTarget {
  void terminate();

  void postMessage(any message, optional sequence<Transferable> transfer);
  [TreatNonCallableAsNull] attribute Function? onmessage;
};
Worker implements AbstractWorker;
```

(Example extracted from the W3C Web Workers specification)

As the aforementioned interface states one can either send just a message or send a message with a sequence of transferable objects. From section **2.7.5** of the HTML Standard [14]: *"Some objects support being copied and closed in one operation. This is called transferring the object, and is used in particular to transfer ownership of unsharable or expensive resources across worker boundaries."*

3.1 Elements

Considering the definition of a `Transferable`, it seems like a good alternative to minimize both T_{et} and T_{es}. Even more so when one considers that otherwise objects are copied using structured cloning(explained in section **2.7.6** of that same standard).

To verify our hypothesis, we put together two benchmarks[1] that aim to verify the difference between invoking `postMessage` with and without structured cloning for a `SharedTypedArray`. Both transfer a `SharedTypedArray` back and forth between the UI thread and a worker; the only difference between the two is that one has 100000 (a hundred thousand) elements in the `SharedTypedArray` and the other one 1000000 (a million). As it can be seen from the results in Tab. 2, increasing the amount of elements by 10x does not change the amount of operations that can be performed when using `Transferable` objects, it scales. On the other hand, the amount of operations that can be performed with structured cloning greatly decreases.

For that reason our library only works with `SharedTypedArray`s and `TypedArrays`.

Computer	Mac Book Air
Processor	Intel Core i5
Clock Frequency	1.8 GHz
System Memory	4 GB
Operating System	OS X Yosemite 10.10.3
Browser version	Firefox Nightly 40.0a1 (2015-04-26)

Table 1. Benchmarks environment.

[1] All benchmarks in this document use the environment described in Tab. 1.

Elements	Cloning [ops/sec]	Transferring [ops/sec]
$1E5$	972	**1705**
$1E6$	114	**754**

Table 2. Difference between transferring and cloning shared buffers.

3.2 Functions

The library must handle the distribution of the code between the different workers when an operation is executed. This is a challenging problem with a lot of possible alternatives [15] [16].

A naive approach would be to try to transfer a function as part of the `message` parameter of `postMessage` but that is not an option (it throws an error).

Considering the fact that `Function` objects cannot be directly transferred a different serialization approach is to be considered (Sec. **??** proposes a solution to sharing additional data and functions between the UI thread and workers) a possible alternative is to:

1. Serialize the `Function` to a `String`.
2. Transfer the `String` to the worker.
3. Create a new `Function` on the worker from that `String`.

Serialization and deserialization One possible way of passing serialized `Function` to workers would be to encode them as binary and transfer them as an `ArrayBuffer`. To encode the strings there are two possible approaches:

– Using the Encoding API[17]
– Implementing non native encoding/decoding functions

The benchmarks provided the following results:

Native encoding API [ops/sec]	Non-native function [ops/sec]
5247	4685

Table 3. Difference encoding/decoding strings natively and in JavaScript.

Transference `Function`'s code not only needs to be serialized but also transferred. For that reason it was also worth comparing the time it takes to encode/decode a `String` and tranfer the resulting `ArrayBuffer` to the worker against just passing the `String` to the worker.

Additionally, in most scenarios code would have to be sent along with a `TypedArray` so it would also be interesting to see if the transfer time was affected by this fact.

We created a benchmark using two functions whose string representation has different lengths to see if the function's length affected the serialization and transfer time (results in Tab. 4). We measured three approaches to transfer the string:

1. Encoding the `String` into an `TypedArrayBuffer`, sending it as a `Tranferable` and decoding it on the Web Worker.
2. Sending the `String` directly.
3. Using a `Blob` to store the stream and retrieving that in the Worker.

Function length	Blob [ops/sec]	Copy [ops/sec]	Tranferable [ops/sec]
Short	1251	**7046**	6219
Long	705	1176	**1198**

Table 4. Sending function strings from UI thread to Web Workers.

We also created a benchmark that tranfers a `SharedTypedArray` and a `String` that are properties of the same `message` object back and forth between the UI thread an a Web Worker to understand if transferring additional objects affected the transfer time when comparing it with just tranferring a `String` (results in Tab. 5.

Copying code [ops/sec]	Tranferring code [ops/sec]
4200	**4340**

Table 5. Sending code and a TypedArray to a Web Worker.

4 Paper Preparation

Springer provides you with a complete integrated LaTeX document class (`llncs.cls`) for multi-author books such as those in the LNCS series. Papers not complying with the LNCS style will be reformatted. This can lead to an increase in the overall number of pages. We would therefore urge you not to squash your paper.

Please always cancel any superfluous definitions that are not actually used in your text. If you do not, these may conflict with the definitions of the macro package, causing changes in the structure of the text and leading to numerous mistakes in the proofs.

If you wonder what LaTeX is and where it can be obtained, see the "*LaTeX project site*" (`http://www.latex-project.org`) and especially the webpage "*How to get it*" (`http://www.latex-project.org/ftp.html`) respectively.

When you use LaTeX together with our document class file, `llncs.cls`, your text is typeset automatically in Computer Modern Roman (CM) fonts. Please do *not* change the preset fonts. If you have to use fonts other than the preset fonts, kindly submit these with your files.

Please use the commands `\label` and `\ref` for cross-references and the commands `\bibitem` and `\cite` for references to the bibliography, to enable us to create hyperlinks at these places.

For preparing your figures electronically and integrating them into your source file we recommend using the standard LaTeX `graphics` or `graphicx` package. These provide the `\includegraphics` command. In general, please refrain from using the `\special` command.

Remember to submit any further style files and fonts you have used together with your source files.

Headings. Headings should be capitalized (i.e., nouns, verbs, and all other words except articles, prepositions, and conjunctions should be set with an initial capital) and should, with the exception of the title, be aligned to the left. Words joined by a hyphen are subject to a special rule. If the first word can stand alone, the second word should be capitalized.

Here are some examples of headings: "Criteria to Disprove Context-Freeness of Collage Language", "On Correcting the Intrusion of Tracing Non-deterministic Programs by Software", "A User-Friendly and Extendable Data Distribution System", "Multi-flip Networks: Parallelizing GenSAT", "Self-determinations of Man".

Lemmas, Propositions, and Theorems. The numbers accorded to lemmas, propositions, and theorems, etc. should appear in consecutive order, starting with Lemma 1, and not, for example, with Lemma 11.

4.1 Figures

For LaTeX users, we recommend using the *graphics* or *graphicx* package and the `\includegraphics` command.

Please check that the lines in line drawings are not interrupted and are of a constant width. Grids and details within the figures must be clearly legible and may not be written one on top of the other. Line drawings should have a resolution of at least 800 dpi (preferably 1200 dpi). The lettering in figures should have a height of 2 mm (10-point type). Figures should be numbered and should have a caption which should always be positioned *under* the figures, in contrast to the caption belonging to a table, which should always appear *above* the table; this is simply achieved as matter of sequence in your source.

Please center the figures or your tabular material by using the `\centering` declaration. Short captions are centered by default between the margins and typeset in 9-point type (Fig. 1 shows an example). The distance between text and figure is preset to be about 8 mm, the distance between figure and caption about 6 mm.

To ensure that the reproduction of your illustrations is of a reasonable quality, we advise against the use of shading. The contrast should be as pronounced as possible.

If screenshots are necessary, please make sure that you are happy with the print quality before you send the files.

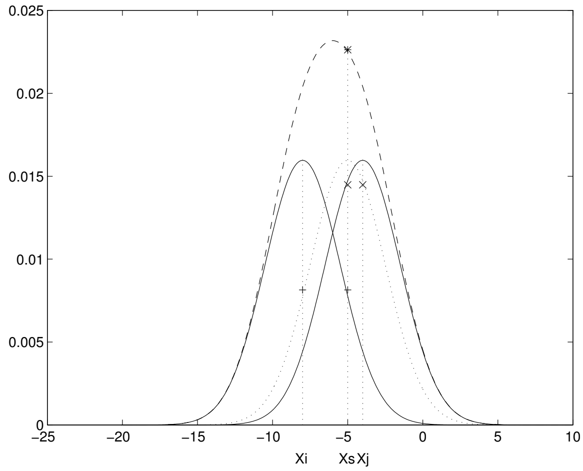

Fig. 1. One kernel at x_s (*dotted kernel*) or two kernels at x_i and x_j (*left and right*) lead to the same summed estimate at x_s. This shows a figure consisting of different types of lines. Elements of the figure described in the caption should be set in italics, in parentheses, as shown in this sample caption.

Please define figures (and tables) as floating objects. Please avoid using optional location parameters like "`[h]`" for "here".

Remark 1. In the printed volumes, illustrations are generally black and white (halftones), and only in exceptional cases, and if the author is prepared to cover the extra cost for color reproduction, are colored pictures accepted. Colored pictures are welcome in the electronic version free of charge. If you send colored figures that are to be printed in black and white, please make sure that they really are legible in black and white. Some colors as well as the contrast of converted colors show up very poorly when printed in black and white.

4.2 Formulas

Displayed equations or formulas are centered and set on a separate line (with an extra line or halfline space above and below). Displayed expressions should be numbered for reference. The numbers should be consecutive within each section or within the contribution, with numbers enclosed in parentheses and set on the right margin – which is the default if you use the *equation* environment, e.g.,

$$\psi(u) = \int_o^T \left[\frac{1}{2}\left(\Lambda_o^{-1}u, u\right) + N^*(-u)\right] dt \; . \tag{5}$$

Equations should be punctuated in the same way as ordinary text but with a small space before the end punctuation mark.

4.3 Footnotes

The superscript numeral used to refer to a footnote appears in the text either directly after the word to be discussed or – in relation to a phrase or a sentence – following the punctuation sign (comma, semicolon, or period). Footnotes should appear at the bottom of the normal text area, with a line of about 2 cm set immediately above them.[2]

4.4 Program Code

Program listings or program commands in the text are normally set in typewriter font, e.g., CMTT10 or Courier.

Example of a Computer Program

```
program Inflation (Output)
  {Assuming annual inflation rates of 7%, 8%, and 10%,...
   years};
   const
     MaxYears = 10;
   var
     Year: 0..MaxYears;
     Factor1, Factor2, Factor3: Real;
   begin
     Year := 0;
     Factor1 := 1.0; Factor2 := 1.0; Factor3 := 1.0;
     WriteLn('Year  7% 8% 10%'); WriteLn;
     repeat
       Year := Year + 1;
       Factor1 := Factor1 * 1.07;
       Factor2 := Factor2 * 1.08;
       Factor3 := Factor3 * 1.10;
       WriteLn(Year:5,Factor1:7:3,Factor2:7:3,Factor3:7:3)
     until Year = MaxYears
end.
```

(Example from Jensen K., Wirth N. (1991) Pascal user manual and report. Springer, New York)

[2] The footnote numeral is set flush left and the text follows with the usual word spacing.

4.5 Citations

For citations in the text please use square brackets and consecutive numbers: [18], [19], [21] – provided automatically by LaTeX's `\cite` ... `\bibitem` mechanism.

4.6 Page Numbering and Running Heads

There is no need to include page numbers. If your paper title is too long to serve as a running head, it will be shortened. Your suggestion as to how to shorten it would be most welcome.

5 LNCS Online

The online version of the volume will be available in LNCS Online. Members of institutes subscribing to the Lecture Notes in Computer Science series have access to all the pdfs of all the online publications. Non-subscribers can only read as far as the abstracts. If they try to go beyond this point, they are automatically asked, whether they would like to order the pdf, and are given instructions as to how to do so.

Please note that, if your email address is given in your paper, it will also be included in the meta data of the online version.

6 BibTeX Entries

The correct BibTeX entries for the Lecture Notes in Computer Science volumes can be found at the following Website shortly after the publication of the book: `http://www.informatik.uni-trier.de/~ley/db/journals/lncs.html`

Acknowledgments. The heading should be treated as a subsubsection heading and should not be assigned a number.

7 The References Section

In order to permit cross referencing within LNCS-Online, and eventually between different publishers and their online databases, LNCS will, from now on, be standardizing the format of the references. This new feature will increase the visibility of publications and facilitate academic research considerably. Please base your references on the examples below. References that don't adhere to this style will be reformatted by Springer. You should therefore check your references thoroughly when you receive the final pdf of your paper. The reference section must be complete. You may not omit references. Instructions as to where to find a fuller version of the references are not permissible.

We only accept references written using the latin alphabet. If the title of the book you are referring to is in Russian or Chinese, then please write (in Russian) or (in Chinese) at the end of the transcript or translation of the title.

The following section shows a sample reference list with entries for journal articles [18], an LNCS chapter [19], a book [20], proceedings without editors [21] and [22], as well as a URL [23]. Please note that proceedings published in LNCS are not cited with their full titles, but with their acronyms!

References

1. Moore, G.E.: Cramming More Components onto Integrated Circuits. (1965)
2. Parallel Programming in the .NET Framework, Microsoft Corp., `http://msdn.microsoft.com/en-us/library/dd460693(v=vs.110).aspx`
3. EcmaScript, ECMA, `http://www.ecmascript.org/`
4. asm.js, `http://asmjs.org/`
5. WebGL, Khronos Group, `http://www.khronos.org/webgl/`
6. Herhut, S., Hudson, R. L., Shpeisman, T., Sreeram, J.: River trail: A path to parallelism in javascript. SIGPLAN Not., 48(10):729–744, (2013).
7. Wang, J., Rubin, N., Yalamanchili, S.: River trail: ParallelJS: An Execution Framework for JavaScript on Heterogeneous Systems. SIGPLAN Not. (2014)
8. McCutchan, J., Feng, H., Matsakis, N. D., Anderson, Z., Jensen, P.: A SIMD Programming Model for Dart, JavaScript, and other dynamically typed scripting languages (2014)
9. Web Workers, W3C, `http://www.w3.org/TR/workers/`
10. Concurrency model and Event Loop, Mozilla Developer Network, `https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop`
11. v8, Google, `https://developers.google.com/v8/intro`
12. SpiderMonkey, Mozilla, `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`
13. Schmidt, D. C., Vinoski, S.,: Comparing Alternative Programming Techniques for Multi-threaded CORBA Servers: Thread Pool (1996)
14. HTML Standard, WHATWG, `https://html.spec.whatwg.org`
15. Epstein, J., Black, A. P., Peyton-Jones, S.: Towards Haskell in the Cloud (2011)
16. Schwendner, A.: Distributed Functional Programming in Scheme (2010)
17. Encoding Standard, WHATWG, `https://encoding.spec.whatwg.org`
18. Smith, T.F., Waterman, M.S.: Identification of Common Molecular Subsequences. J. Mol. Biol. 147, 195–197 (1981)
19. May, P., Ehrlich, H.C., Steinke, T.: ZIB Structure Prediction Pipeline: Composing a Complex Biological Workflow through Web Services. In: Nagel, W.E., Walter, W.V., Lehner, W. (eds.) Euro-Par 2006. LNCS, vol. 4128, pp. 1148–1158. Springer, Heidelberg (2006)
20. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure. Morgan Kaufmann, San Francisco (1999)
21. Czajkowski, K., Fitzgerald, S., Foster, I., Kesselman, C.: Grid Information Services for Distributed Resource Sharing. In: 10th IEEE International Symposium on High Performance Distributed Computing, pp. 181–184. IEEE Press, New York (2001)
22. Foster, I., Kesselman, C., Nick, J., Tuecke, S.: The Physiology of the Grid: an Open Grid Services Architecture for Distributed Systems Integration. Technical report, Global Grid Forum (2002)
23. National Center for Biotechnology Information, `http://www.ncbi.nlm.nih.gov`

Appendix: Springer-Author Discount

LNCS authors are entitled to a 33.3% discount off all Springer publications. Before placing an order, the author should send an email, giving full details of his or her Springer publication, to `orders-HD-individuals@springer.com` to obtain a so-called token. This token is a number, which must be entered when placing an order via the Internet, in order to obtain the discount.

8 Checklist of Items to be Sent to Volume Editors

Here is a checklist of everything the volume editor requires from you:

- [ ] The final LaTeX source files
- [ ] A final PDF file
- [ ] A copyright form, signed by one author on behalf of all of the authors of the paper.
- [ ] A readme giving the name and email address of the corresponding author.