Multi-Newton: A Multi-Core Benchmark Used to Improve Algorithmic Differentiation

Bradley M. Bell and Peter Hepperger
October 1, 2012

***Abstract*—Implementing code that is efficient in a multi-core environment is one of the challenges of modern software engineering. To meet this challenge, future software, including Automatic Differentiation (also called Algorithmic Differentiation and abbreviated as AD) will have to avoid excessive synchronization barriers between threads as well as manage memory efficiently. We present an application of Newton's method that finds all the zeros of a nonlinear function in an interval. This is being used as a benchmark to improve the multi-core performance of the open source AD software package CppAD. Different threading systems, and different versions of this software package, are easily compared for speed of execution. As an example of the use of this benchmark, we compare the results for three versions of the software package. These versions correspond to improvements in a general purpose C++ multi-threading memory allocator. The benchmark, and the multi-threaded memory allocator, are distributed with CppAD.**

***Index Terms*—benchmark, multi-thread, multi-core, memory allocation, OpenMP, pthreads, boost-threads, automatic differentiation, AD**

I. Introduction

Increasing the number of CPU cores has replaced increasing clock speed as a means of making computers faster [1]. The advent of inexpensive computers with many cores makes it important to take advantage of these cores; e.g., [2]. This requires substantial changes to software, including Automatic Differentiation (AD); see Section V or [3]. Reverse mode AD uses a single global tape that records the floating point operations corresponding to a function evaluation; e.g., [4]. This approach is not thread-safe because different threads would be reading and writing the same memory.

Introducing separate tapes for each thread poses additional challenges even if no explicit barriers or synchronization occur. Allocating memory causes implicit blocking while threads wait a substantial amount of time for other threads before their request for memory is finally executed by the operating system. Benchmark testing made this problem evident in version 2011 of CppAD [5]. A special memory allocator, that is built on top of the system allocator, is now used to avoid this problem. For a discussion of other multi-threading memory allocators see [6] and [7] or search the web for `mtmalloc`, `tcmalloc`, and `pthread_alloc`. A special allocator was written because this AD software makes extensive use of static memory and efficient management of this memory is key to its performance. In addition, the special allocator can be turned off and any replacement for the system allocator can be used.

Besides the memory allocation, managing a team of threads causes further overhead, which may not amortize for small tasks. This includes increased inter-processor communication due to invalid cache pages caused by writing to shared arrays and data structures, often referred to as false sharing; e.g., [8].

Testing the actual speed of parallel algorithms is an important step in the development of multi-threaded software; see, [9], [10], [11], [12]. In this paper, we describe the details of a multi-threaded Newton method benchmark that can be used with and without AD. It is important to compare results for different threading systems; e.g., [13]. The benchmark code that is particular to the threading system is separate and has an application interface (API) that is easy to implement for a particular threading library. We show the benchmark results for various machines with 48 cores, using various threading systems, and various versions of the open source AD software package CppAD. The benchmark, and the multi-threaded memory allocator, are distributed with this package.

II. Benchmark System

In order to obtain reliable results, tests with a runtime of less than one second are repeated until a runtime of at least one second is reached and the average runtime is used for the corresponding test results. The routine `team_create`, called at the beginning of the program, creates new threads. The routine `team_destroy`, called at the end of the program, removes the new threads. The runtime for the create and destroy routines is not included in the average runtimes. The routine `thread_work` is called once for each repeat of the test and the runtime for this routine is included in the average runtimes. The time used to divide the work up, before each call to `thread_work`, and to combine the results, after each call to `thread_work`, is also included. The average runtime is used for our benchmark comparisons, see, e.g., the results in Figure 2.

The multi-threading benchmark tests are the same for each threading system. The only threading system specific code is an implementation of the `team_thread` interface specified below. The following functions are used in this specification:

m = `thread_num()`
: The return value *m* has type `size_t` and uniquely identifies the current thread. This value is between zero and the total number of threads minus one.

b = `in_parallel()`
: The return value *b* has type `bool`. If the current

B.M. Bell is with IHME & APL University of Washington, Seattle, Washington USA, `bradbell@uw.edu`

P. Hepperger is with Technische Universität München, Garching bei München, Germany, `peter.hepperger@tum.de`

execution mode is known to be sequential, *b* is `false`. Otherwise it is `true` (current mode may be parallel).

The following threading specific interface is declared in the file `team_thread.hpp`. These routines start, use, and stop a team of threads that can be used with the benchmark test. Example implementations for OpenMP threads, Boost threads, and POSIX threads are provided with the CppAD package.

The return value *ok* has type `bool` and is `false` if an error is detected, otherwise it is `true`. Calls to the routines `team_create`, `team_work`, and `team_destroy`, can only be done by the master thread; i.e., the thread number must be zero. In addition, they can only be done in sequential execution mode with *in_parallel*`()` equal to `false`. Execution will also be sequential when these calls return.

ok = `team_create(` *num_threads* `)`
: This call creates a team of threads. The argument *num_threads* has type `size_t`, is greater than 0, and specifies the number of threads in this team. This initializes `team_work` to be used with *num_threads* threads. It also creates *num_threads*-1 new threads and puts them in a waiting state.

ok = `team_work(` *worker* `)`
: This routine may be called one or more times between the call to `team_create` and `team_destroy`. The argument *worker* is a function pointer of type `bool` *worker*`(void)`. Each call to `team_work` runs *num_threads* versions of *worker* with the corresponding `thread_num()` between 0 and *num_threads*-1 and different for each thread.

ok = `team_destroy()`
: This routine terminates all the threads except for the master; i.e., it terminates the threads corresponding to thread number *m* = 1 , ... , *num_threads*-1.

III. Multi-Threaded Newton Method Benchmark

A. Bounded Newton Method

Given lower and upper bounds $[a, b]$, a maximum number of Newton iterations K, and a convergence criterion ε, the bounded Newton method either returns a singleton $\{x_k\}$ such that $x_k \in [a, b]$ and $|f(x_k)| \leq \varepsilon$ or it returns the empty set $\emptyset$.

1) Set $x_0 = (a + b)/2$, $k = 0$.
2) If $|f(x_k)| < \varepsilon$, return $\{x_k\}$.
3) If $k = K$, return $\emptyset$.
4) If $f(x_k)f^{(1)}(x_k) \geq 0$ and $x_k = a$, return $\emptyset$.
5) If $f(x_k)f^{(1)}(x_k) \leq 0$ and $x_k = b$, return $\emptyset$.
6) Set $y_k = x_k - f(x_k)/f^{(1)}(x_k)$.
7) Set $x_{k+1} = \min[b\ ,\ \max(a\ ,\ y_k)]$, $k = k + 1$, Goto 2.

B. Multi-Threaded Newton Method

Given lower and upper bounds $[\alpha, \beta]$, a maximum number of Newton iterations K, a convergence criterion ε, a number of threads M, and a number of sub-intervals J, the multi-threaded Newton method runs the bounded Newton method on J equally spaced sub-intervals of $[\alpha, \beta]$ and returns the union of all of the solution points.

1) Sequential Division of Work: For $m = 0, \ldots, M - 1$, denote the number of sub-intervals for thread m by

$$L[m] = \begin{cases} \text{floor}(J/M) + 1 & \text{if } m < \text{mod}(J, M), \\ \text{floor}(J/M) & \text{otherwise.} \end{cases}$$

Use $\gamma = (\beta - \alpha)/J$ to denote the length of each of the J sub-intervals. The master thread computes $s[m]$ (the start) and $e[m]$ (the end) of the sub-intervals handled by thread m where

$$s[m] = \alpha + \gamma \sum_{p<m} L[p] \ , \quad e[m] = s[m] + \gamma L[m] \ .$$

These are recomputed for each repeat of this test. Note, for $m > 0$, $s[m] = e[m - 1]$.

2) Parallel Computation: For thread $m = 0, \ldots, M - 1$, the following computations are done in parallel: For $\ell = 0, \ldots, L[m]$, the lower and upper bound of the ℓ-th sub-interval for thread m are

$$a[m, \ell] = s[m] + \gamma\ell \ , \quad b[m, \ell] = e[m] - \gamma(L[m] - 1 - \ell).$$

Note that $a[m, 0] = s[m]$ and $b[m, L[m] - 1] = e[m]$ (even with round-off error in floating point arithmetic). The bounded Newton method is applied on the sub-intervals $[\ a[m, \ell]\ ,\ b[m, \ell]\ ]$. Use $S[m]$ to denote the union from $\ell = 0$ to $\ell = L[m] - 1$ of the bounded Newton results; i.e., the approximate solutions for thread m. This set is stored as a vector in increasing order (to make the sequential combination of work faster). For $\ell > 0$ it is necessary to check that the solution is not the same as the solution for the previous sub-interval. This can happen when the absolute function value is less that ε at the lower bound for this interval (upper bound for the previous interval). If the distance between two solutions is less that γ, the solutions are combined to be the one corresponding to the smaller absolute function value.

3) Sequential Combination of Work: The master thread computes the union from $m = 0$ to $m = M - 1$ of $S[m]$ and returns it as the set of approximate solutions for the entire interval $[\alpha, \beta]$. For each $m > 0$, it is necessary to check that the first solution in $S[m]$ is not within γ of the last solution in $S[m - 1]$, which can happen if one of them is equal to $s[m] = e[m - 1]$.

IV. Sine Wave Example

For a concrete example, we apply the multi-threaded Newton method to find zeros of the function

$$f(x) = \frac{1}{N} \sum_{n=0}^{N-1} \sin(x).$$

The parameter N controls how many sine function evaluations and binary sums are performed during each function evaluation. For each iteration of the bounded Newton method, the derivative $f^{(1)}(x_k)$ is given by

$$f^{(1)}(x_k) = \frac{1}{N} \sum_{n=0}^{N-1} \cos(x_k). \tag{1}$$

The multi-threaded Newton method lower and upper bounds are $\alpha = 0$, $\beta = (Z - 1)\pi$, where $Z = 500$ is the number

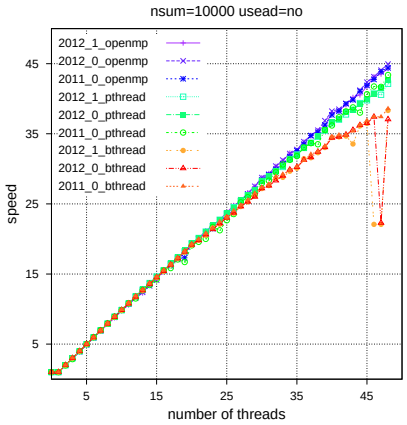

Fig. 1. Hand coded derivative results for all versions and all threadings using 48 cores and $N = 10^4$.

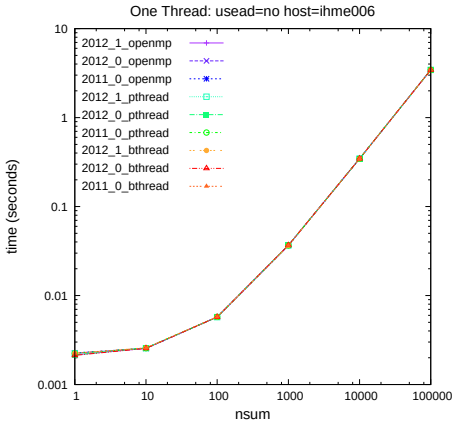

Fig. 2. Hand coded derivative results for all versions, all threadings, using one thread and the same host for all cases (times are nearly identical for all nine cases).

of zeros in the interval $[\alpha, \beta]$. The number of sub-intervals $J = 5000$, the tolerance ε is 10^2 times the upper bound β times double precision machine epsilon, and the maximum number of iterations is $K = 20$.

A. Without Algorithmic Differentiation

The results in this section use equation (1), to compute the derivatives required by the Bounded Newton Method; see Section III-A. The results for versions `2011_0`, `2012_0` and `2012_1`, using OpenMP, Posix, and Boost threading with $N = 10^4$ are plotted in Figure 1. The number of threads is plotted on the horizontal axis and the corresponding speed (average execution time for one thread divided by the average execution time for the number of threads) is plotted on the vertical axis. The number of subintervals for thread m is floor(J/M) or floor$(J/M) + 1$. Since the number of threads $M \leq 48$ and $J = 5000$, it follows that $J/M > 100$ and hence each thread is doing about the same amount of work.

Each curve corresponds to one machine and the machine chosen was from one of two types. One type of machine had 48 AMD Opteron(tm) 6180 SE processors (512 KB cache and 2500 MHz). The other type had 64 AMD Opteron(tm) 6262 HE processors (2048 KB cache and 1600 MHz). Scaling each curve by the time for one thread corrects for the effect or using different machines. (Our tests indicate that these scaled values are nearly the same for the two types of machines used.) The horizontal axis value zero corresponds to one thread, but without use of the specific threading system.

For each curve in Figure 1, the speed difference between horizontal axis values zero and one compares the different threading systems to no threading system. While this is an absolute comparison of the normalization (time for one thread) between different threading systems, it is not an absolute comparison between the different versions. Figure 2 plots the absolute execution time, corresponding to one thread as a function of N and for a fixed host machine. The horizontal axis value $N = 10^4$ compares the normalization factors (time for one thread) used in the previous figure. Note that these absolute times are almost identical for the different versions and threading systems and that they are approximately linear with respect to N for $N \geq 10^2$. Also note that, for the previous figure, the value of N divided by the maximum number of threads is greater than this value; i.e., $10^4/48 > 10^2$.

Using multiple threads requires additional work for thread management. The time to create and destroy threads is not included in our results. On the other hand, time to split up the work, wake up the other threads, put the threads back to sleep, and combine the work, is included in our test results; for more details, see the discussion at the beginning of Section II. While threading libraries provide a way to minimize this overhead, it is never negligible. In addition, false sharing has a major impact on efficiency in a multi-threading environment; see [8].

It is interesting to see the difference in performance between the different threading systems when $N = 10^3$, and hence there is less work per thread; see Figure 3. Note that OpenMP threading is doing the best, while Posix threading and Boost threading seem to have an overhead that is significant for this size of problem. This difference may be related to the implementation of the threading independent interface; see `team_create`, `team_work`, and `team_destroy` in Section II. The OpenMP system has higher level primitives that make this interface simpler to implement. The implementations for both Posix and Boost threading are similar. They use thread local storage to implement `thread_num()`, and barriers to implement `team_work()`. The actual implementations can be found in [5]. It is also interesting to note that the performance for Posix and Boost threading starts to degrade at 15 threads and $10^3/15 < 10^2$. Hence the amount of work per thread is in the non-linear region of the plot in Figure 2.

B. Using Algorithmic Differentiation

The results in this section use Algorithmic Differentiation to compute the derivatives required by the Bounded Newton Method. The results for versions `2011_0`, `2012_0` and `2012_1`, using OpenMP, Posix, and Boost threading with $N = 10^2$ and $N = 10^3$ are plotted in Figures 4 and 5. These results show that, for all threadings, version `2011_0` is slower that version `2012_0` and `2012_0` is slower than `2012_1`. The difference between these versions is discussed in Section VI.

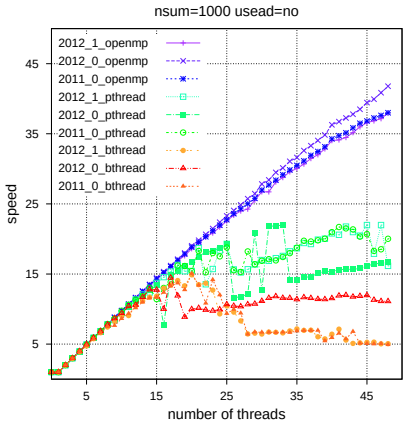

Fig. 3. Hand coded derivative results for all versions and all threadings using 48 cores and $N = 10^3$.

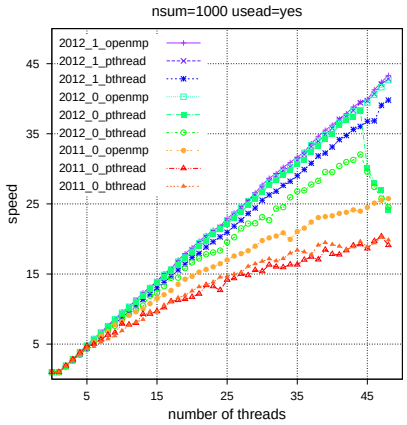

Fig. 5. AD derivative results for all versions and all threadings using 48 cores and $N = 10^3$.

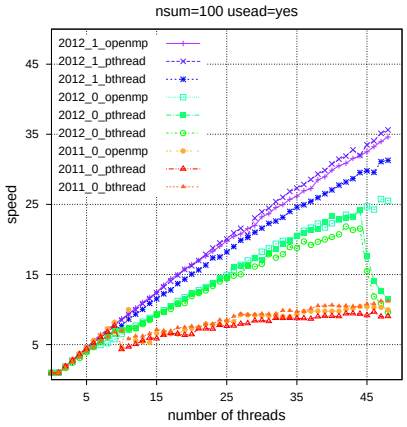

Fig. 4. AD derivative results for all versions and all threadings using 48 cores and $N = 10^2$.

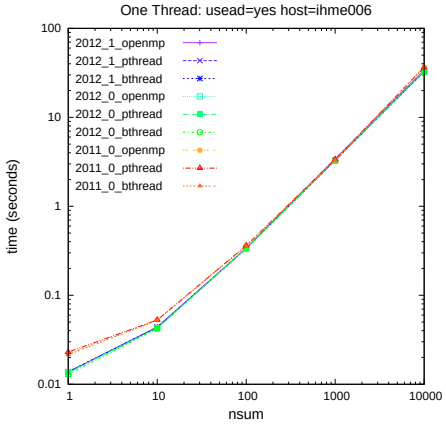

Fig. 6. AD derivative results for all versions and all threadings using one thread and the same host for all cases.

This improvement would not have been possible without the benchmark presented in this paper. The corresponding absolute times for one thread are plotted in Figure 6. Note that, for $N = 10^2$ and $N = 10^3$, these absolute times are almost identical for the different versions and threading systems. In addition, the time is a linear function of N between $N = 10$ and $N = 10^4$. Perhaps this is why the performance in Figure 4 starts to degrade when 10^2 divided by the number of threads is less than 10 (because the work per thread is too small for optimal use of the different threads).

V. Taped Algorithmic Differentiation and Memory Usage

CppAD records the floating point operations corresponding to an algorithm. It then uses this recording to computes arbitrary order forward and reverse mode derivatives in a manner inspired by the software package ADOL-C [14]. This is accomplished by overloading the floating point operators (e.g., division) and elementary mathematical functions (e.g., the sine function). We refer to these overloaded operators and functions as atomic operations which are actually functions with one or two arguments and one result. First the independent variables are identified by the user. Then each atomic operation, with arguments that depends on the independent variables, creates a new variable corresponding to the result. The overloaded versions of the atomic operations stores the operations that depend on the independent variables in a recording (often referred to as the tape). One difference from ADOL-C is that the type for the floating point operations is a template parameter. This enables one to use AD to differentiate a function where the source code that defines the function also uses AD (using `AD<AD<double>>`). It also enables many other applications; e.g., using interval arithmetic for the floating point type and getting bounds on derivatives.

In the case of the sine wave example in Section IV there are approximately $2N$ atomic operations. When an operation recording gets too long for the current capacity of the vectors storing the recording, it is extended and copied to new vectors (which involves additional memory allocation). The sine function is special in that, although only one variable per operation is visible to the user, an extra hidden variable is created for each operation. This variable corresponds to the cosine function (because the derivative of the sine is the cosine and the derivative of the cosine is minus the sine). The values and derivatives for many of the standard math functions are often computed in pairs. After the tape is recorded, the

function values corresponding to each operation are allocated and stored. Then the derivative values corresponding to each operation are allocated and stored. (One can preallocate space for the derivative values, and one can convert the entire summation to a single operator, but this is not done by the benchmark.)

VI. The CppAD Multi-Threaded Memory Allocator

This multi-threading memory allocator `thread_alloc` comes with version 2012 and can be used separately; i.e., without including all of CppAD. This allocator returns memory with certain, predefined, discrete capacities in terms of bytes of memory. We denote these capacities by $c_i \in \mathbf{Z}_+$ for $i = 0, ..., I-1$, where $c_0 = 128$ and

$$c_{i+1} = 3 * \text{floor}[(c_i + 1)/2].$$

The capacity values are not part of the CppAD API because they are subject to future changes (as implemented, it is easy to change them). When a memory request is made, both a pointer to the memory, and the corresponding capacity is returned. To be specific, if b bytes of memory are requested, a pointer to c_i bytes, and the value c_i are returned, where i is the minimal index such that $b \leq c_i$. Containers (e.g. `CppAD::vector`) can use the capacity information to determine if there is enough space to add more elements or if another call to the memory allocator must be made.

Special versions were created for this paper using the script `bin/special_version.sh` which comes with CppAD [5]. This simplifies what is different between the versions and enables others to reproduce the results in this paper. **Version 2011_0** uses the system allocator with the capacity modification mentioned above. In order to be thread-safe, the system memory allocator uses mutual exclusion; i.e., only one thread at a time can allocate new memory. When there are lots of threads and memory allocation requests, a significant amount of time is spent while threads wait for their turn.

To avoid this sort of blocking, memory that is returned to **Version 2012_0** of `thread_alloc` is held for future use by the same thread and not returned to the system immediately. This is accomplished using a singly linked list of these available blocks of memory. For thread $m = 0, \ldots, M-1$, and capacity index $i = 0, ..., I-1$, there is a root pointer to the list of available blocks which we denote by $A_{m \cdot I + i}$. When a block is returned to `thread_alloc`, it is placed at the front of this list. When a memory block of capacity c_i is requested for thread m, it is taken from the front of this list. In the special case where there is no previously allocated memory available, the system allocator is used to get more memory for this thread. A special function call that returns all allocated memory to the system, is also provided.

In version 2012_0, the roots of the linked lists $A_{m \cdot I + i}$ are stored in in a single vector of contiguous memory. When a thread changes one of these values, it invalidates the cache the other threads are using to access this vector. This is referred to as *false sharing*, since no two threads actually modify the same data; see [8]. In addition, version 2012_0 also has two vectors of length M that are used to count the number of bytes that are currently in use and available for each thread. These vectors also have the false sharing problem described above.

In **Version 2012_1**, a structure is allocated separately for each thread. This contains the counters for the number of bytes that are currently in use and available for a thread. It also contains two vectors of length I. One vector holds the available list root pointer for each capacity and this thread. The other vector holds the in use root pointer for each capacity and this thread (this is only used by the debug version of the program). If a thread changes its structure, the cache for other threads is still valid because the structure for other threads is stored in a different memory area.

You can create one of the versions compared in this paper by executing the following commands (where `<version>` is replaced by `2011_0`, `2012_0`, or `2012_1`):

```
svn checkout \
https://projects.coin-or.org/svn/CppAD/trunk
cd trunk ; bin/special_version.sh <version>
```

References

[1] S. Pllana and J. L. Traff, "Introduction to the Scientific Programming special issue: Software development for multi-core computing systems," *Scientific Programming*, vol. 17, no. 4, pp. 283–284, 2009.

[2] Y. Cui, Y. Wang, Y. Chen, and Y. Shi, "Experience on comparison of operating systems scalability on the multi-core architecture," *IEEE International Conference on Cluster Computing*, pp. 205–215, 2011.

[3] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation, Second Edition*. SIAM, 2008.

[4] M. Fagan and A. Carle, "Reducing reverse-mode memory requirements by using profile-driven checkpointing," *Future Generation Computer Systems*, vol. 21, no. 8, pp. 1380–1390, 2005.

[5] B. Bell, *CppAD: a package for C++ algorithmic differentiation*. http://www.coin-or.org/CppAD: Computational Infrastructure for Operations Research, 2012.

[6] E. Berger, K. McKinley, R. Blumore, and P. Wilson, "Hoard: a scalable memory allocator for multithreaded applications," *Operating Systems Review*, vol. 34, no. 5, pp. 117–28, 2000.

[7] S. Schneider, D. Christos, and D. Nikolopoulos, "Locality-conscious multithreaded memory allocation," *International Symposium on Memory Management*, vol. 2006, pp. 84–94, 2006.

[8] T. Liu and E. D. Berger, "SHERIFF: precise detection and automatic mitigation of false sharing," *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011.

[9] K.-Y. Chen, J. M. Chang, and T.-W. Hou, "Multithreading in Java: Performance and scalability on multicore systems," *IEEE Transactions on Computers*, vol. 60, no. 11, pp. 1521–34, 2011.

[10] P. Garcia and H. Korth, "Multithreaded architectures and the sort benchmark," *1st International Workshop on Data Management on New Hardware, DaMoN 2005, Co-located with ACM SIGMOD/PODS 2005*, 2005.

[11] C.-S. Koong, C. Shih, P.-A. Hsiung, H.-J. Lai, C.-H. Chang, W. C. Chu, N.-L. Hsueh, and C.-T. Yang, "Automatic testing environment for multi-core embedded software - ATEMES," *Journal of Systems and Software*, vol. 85, no. 1, pp. 43–60, 2012.

[12] N. Zhang, "Computing optimised parallel speeded-up robust features (p-surf) on multi-core processors," *International Journal of Parallel Programming*, vol. 38, no. 2, pp. 138–158, 2010.

[13] P. Kegel, M. Schellmann, and S. Gorlatch, "Comparing programming models for medical imaging on multi-core systems," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 10, pp. 1051–1065, 2011.

[14] A. Griewank, A. Walther, and K. Kulshreshtha, "ADOL-C: automatic differentiation by overloading in C++," *Computational Infrastructure for Operations Research*, vol. coin-or, no. https://projects.coin-or.org/ADOL-C, 2012.