Volume 123, Article No. 123003 (2018) https://doi.org/10.6028/jres.123.003
Journal of Research of National Institute of Standards and Technology

`NISTfit`: *A Natively Multithreaded C++11 Framework for Model Development*

Ian Bell[1] **and Matthias Kunick**[2]

[1]National Institute of Standards and Technology,
Boulder, CO 80301 USA

[2]Zittau/Görlitz University of Applied Sciences
02763 Zittau, Germany

ian.bell@nist.gov
m.kunick@hszg.de

Software DOI: https://doi.org/10.18434/M39W9S

Key words: C++; HPC; optimization.

Accepted: January 24, 2018

Published: February 12, 2018

https://doi.org/10.6028/jres.123.003

1. Summary

The current trend in computer architecture is for increasingly parallel computation while the clock frequency stagnates. The increase in computing speed is achieved by dividing a process into several threads which are executed in parallel on multiple processors, processors with multiple cores, cores that are able to handle multiple threads (hyper-threading), graphical processing units (GPU), or co-processors. In order to take advantage of these new architectures, algorithms that have historically been implemented for serial evaluation need to be refactored for parallelization. In this work, a native multithreading framework in C++11 for scientific and engineering model development is presented.

The motivation for `NISTfit` is to develop a modern C++11-based library for this problem that is:

- Cross-platform: `NISTfit` has only very minimal header-only dependencies (Eigen and ThreadPool), and builds reliably on all major architectures; a `CMake` build file is provided.
- Easy to use: There are a multitude of similar libraries for model fitting (e.g., MINPACK [1], `levmar`[2], Eigen's `LevenbergMarquardt` module [3], to name but a few) that a) are based on archaic FORTRAN/C/C++ constructs, b) require significant boilerplate to solve simple problems, or c) have difficult-to-build dependencies. It is the opinion of the authors of `NISTfit` that `NISTfit` strikes a

[1]http://www.netlib.org/minpack/
[2]http://users.ics.forth.gr/~lourakis/levmar/
[3]https://eigen.tuxfamily.org/dox/unsupported/classEigen_1_1LevenbergMarquardt.html

good balance of power and ease-of-use for simple fitting problems. The code utilizes modern C++11 constructs and will build on any C++11 compliant compiler.

- Parallelizable: The future is parallel, and `NISTfit` is able to achieve near-theoretical speedup as more cores are made available to the fitting for sufficiently expensive models.

2. Software Specifications

NIST Operating Unit	Materials Measurement Lab, Applied Chemistry and Materials Division, Thermophysical Properties of Fluids Group
Category	Optimization, high-performance computing
Operating Systems	Cross-platform
Programming Language	C++11, Python
Inputs/Outputs	Please see sample code.
Disclaimer	https://www.nist.gov/director/licensing

The source code is hosted on GitHub[4], and a zip file of the code as of publication is available in the Supplemental Materials.

3. Problem Statement

To begin, we first need to define the problem that we are trying to solve. In the scheme of least-squares minimization, we have a set of residual functions $\vec{f}$ that are each a residual between a data point and a model prediction, as seen schematically in Fig. 1.

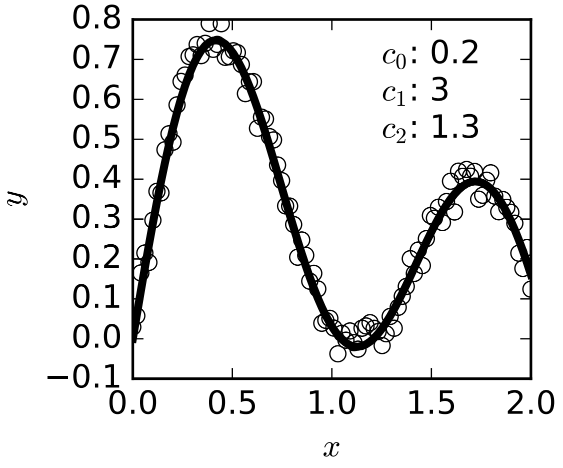


Fig. 1. Example of fitting data for decaying exponential problem (solid line: model, markers: “experimental” data points).

[4]https://doi.org/10.18434/M39W9S

The i-th model prediction is dependent on the array of model parameters $\vec{c}$, or

$$f_i(\vec{c}) = y_{\text{model}}(x_i,\vec{c}) - y_i \tag{1}$$

Our goal then is to find the set of model parameters $\vec{c}$ that best fit the model to the given experimental data. In this case, "best fit" means that we minimize the sum-of-squares error F given by

$$F = \sum_i f_i^2. \tag{2}$$

The Jacobian matrix $\mathbf{J}$ is required for many deterministic optimization methods and is given by

$$\mathbf{J} = \begin{bmatrix} \left(\frac{\partial f_0}{\partial c_0}\right) & \left(\frac{\partial f_0}{\partial c_1}\right) & \cdots & \left(\frac{\partial f_0}{\partial c_m}\right) \\ \left(\frac{\partial f_1}{\partial c_0}\right) & \left(\frac{\partial f_1}{\partial c_1}\right) & \cdots & \left(\frac{\partial f_1}{\partial c_m}\right) \\ \vdots & \vdots & \ddots & \vdots \\ \left(\frac{\partial f_n}{\partial c_0}\right) & \left(\frac{\partial f_n}{\partial c_1}\right) & \cdots & \left(\frac{\partial f_n}{\partial c_m}\right) \end{bmatrix} \tag{3}$$

where in each partial derivative all other model parameters are held constant and where the term $\left(\frac{\partial f_i}{\partial c_j}\right)$ is equivalent to

$$\left(\frac{\partial f_i}{\partial c_j}\right) = \left(\frac{\partial\left[y_{\text{model}}(x_i,\vec{c})\right]}{\partial c_j}\right)_{c_{k\neq j}}. \tag{4}$$

In general, the partial derivatives found in the Jacobian matrix can be evaluated analytically (preferred), or through the use of numerical derivatives if analytic partial derivatives are not available. A class implementing numerical derivatives for the Jacobian matrix is available in `NISTfit`, though it is not further described here.

The key point to note, and the core motivation for the development of `NISTfit`, is that evaluation of the Jacobian matrix and the array of residual functions is an embarrassingly parallel problem. That is to say, each row in $\mathbf{J}$ or residual in $\vec{f}$ is entirely independent of the other ones. Therefore, a library that is able to evaluate $\mathbf{J}$ and $\vec{f}$ in parallel can yield a significant reduction in the time required to carry out one fitting evaluation. Furthermore, the evaluation of $\vec{f}$ and $\mathbf{J}$ represent the building blocks for a wide range of deterministic optimization methods, and the tools developed here could serve as the kernel that is embedded into more advanced optimization tools.

The exercise of solving a system of non-linear equations is mathematically related to the case of least-squares minimization. Fundamentally, they both involve the use of the Jacobian matrix and the vector of residuals to be driven to zero. In the case of Newton-Raphson method, there are as many model parameters as residual functions, unlike the case of the generalized least-square fitting problem in which there are more residual functions than model parameters. The Newton-Raphson method is known to be unreliable for systems of equations that are very nonlinear, though for "simple" problems, the Newton-Raphson system-of-nonlinear-equations solver can yield rapid convergence.

4. C++ Implementation

While there are many programming languages that could be used to develop a flexible and computationally efficient optimization toolbox, C++ was selected in this case due to its computational efficiency, flexibility, and the facility with which it can be integrated into other high level languages like Python. `NISTfit` leverages several of the capabilities introduced in the C++11 standard, most especially native threading.

4.1 Architecture

One of the primary motivations for the use of object-oriented programming languages like C++ is their extensibility. In this sense, extensibility means that the core code with a fixed application programming interface (API) can be used to work on many different problems. Therefore, we discuss here the API for `NISTfit` in some depth so that future users of `NISTfit` might be able to work with the public API. While the public API of `NISTfit` is rather straightforward, tools like `doxygen`[5] can be instrumental in understanding how the pieces of complex C++ programs fit together. To that end, the doxygen-generated documentation for `NISTfit` are provided in the Supplemental Materials, both in HTML and PDF formats.

The serial (one-at-a-time) evaluation of a set of residual functions is a trivial endeavor, whereas efficient parallel evaluation through the use of C++11 threading is much more perilous. Some of the challenges introduced when adding threading are:

- *Initialization*: The initialization and destruction of threads is a non-negligible contribution to the runtime, and should be minimized as much as possible. In `NISTfit`, the threads are initialized at the beginning of the optimization through the use of a ThreadPool[6], the entire optimization is run, and as the `AbstractEvaluator` falls out of scope, the threads are destroyed. Threads are fed their inputs via a queue, and when not carrying out calculations, are sleeping, waiting for a condition variable to be set.
- *Load balancing:* The load between threads should be as balanced as possible so that all the threads start and finish at the same time. As the number of data points increases, the number of points evaluated by each thread by definition approaches parity. Nevertheless, the balancing of the number of evaluations per thread does not necessarily imply that the total amount of work will be balanced across threads. If some outputs are more computationally involved than others, the differences in thread evaluation times can be significant, implying that the faster threads may have to wait an extended time for the slower threads to finish

Two core components form the API of `NISTfit` as shown in Fig. 2: the `AbstractOutput` class, and the `AbstractEvaluator` class. The `AbstractOutput` class contains the output, and the routines for evaluating the model, and `AbstractEvaluator` class is the manager class, and owns the set of classes derived from the `AbstractOutput` class. The evaluator class is also responsible for managing the threads.

4.2 `AbstractOutput` Class

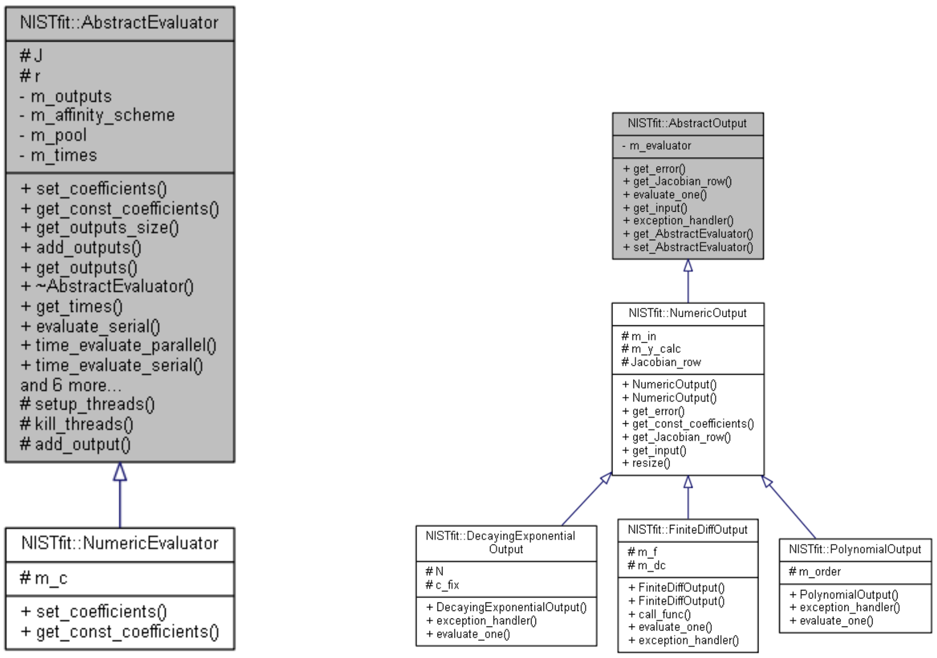
The `AbstractOutput` class (or derived class thereof) carries out the evaluation of the model and the construction of the row in the Jacobian matrix. The `evaluate_one` function is used to carry out the evaluation, and must be implemented for each model, as the example in Sec. 6 demonstrates. The class derived from `AbstractOutput` *must* provide a threadsafe implementation of `evaluate_one`.

4.3 `AbstractEvaluator` Class

The `AbstractEvaluator` class is the core evaluation class of `NISTfit`. This class is used to evaluate all the `AbstractOutput`-derived instances that are owned by the `AbstractEvaluator` class, through the use of serial- or thread-parallel evaluation. A class derived from `AbstractEvaluator` is configured by the following method:

1. an evaluator class deriving from `AbstractEvaluator` is instantiated,

[5]http://www.stack.nl/~dimitri/doxygen/
[6]https://github.com/stfx/ThreadPool2



(a) Inheritance diagram for `AbstractEvaluator`

(b) Inheritance diagram for `AbstractOutput`

Fig. 2. The abstract base classes forming the public API of `NISTfit` (diagrams automatically generated by doxygen).

2. a vector of outputs, each derived from `AbstractOutput`, is passed into the instance via the `add_outputs` function. For each output, a reference is stored to the `AbstractEvaluator` that owns it.

The evaluation of the Jacobian matrix **J** and the residual vector $\vec{f}$ is carried out by:

1. calling either the `evaluate_serial` (for serial evaluation) or `evaluate_parallel` (for thread-parallel evaluation). For each evaluation, the `evaluate_one` function of each `AbstractOutput` is called, thereby calculating the residual vector entry and the row in the Jacobian matrix,

2. retrieving a reference to the Jacobian matrix **J** and vector $\vec{f}$ via the `get_Jacobian_matrix` and `get_error_vector` functions. These matrix-like objects are returned as `const` references in order to avoid copying what can be a rather large amount of data.

4.4 Optimization

As with any application of threading to turn a serial algorithm into a parallel algorithm, the efficiency and speedup when adding thread-level parallelism are dependent on how much work is required to evaluate each element in the array, load balancing (see above), and threading overhead (construction/destruction). In general, the more work required to evaluate one element in the residual array, the more linear the speedup can be as the number of threads is increased because the threading overhead becomes increasingly negligible.

The object-oriented nature of C++ means that if the right architectural decisions are made about the interface at the beginning of the development process, the core code can be extended to solve problems that were not within the scope of the original problem statement. For instance, for `NISTfit`, the code could be used to implement the computational kernel of ODRPACK, generic derivative-based optimization, or other optimization methods.

There are a few key points that drive the efficiency of `NISTfit`:

- Threads are used to evaluate the Jacobian and the array of residuals in parallel.
- Vectorized (or matrix-based) operations are used, delegating to pre-existing tuned linear algebra libraries (`Eigen`).
- The base classes, and their derived classes, are all thread-safe by architecture, though the implementer must be sure to maintain the thread-safety of their implementation.

4.5 Python Wrapper via `pybind11`

The open-source, header-only, library `pybind11` can be used to develop thin wrappers between the C++ code and the Python programming language. The library `pybind11` originated as a fork of the legendary `boost::python` C++/Python interface generator, but has been extended to focus on modern C++ features, and is used in this work to develop 1-to-1 wrappers between the C++ code and Python, one of the eminent open-source languages for scientific computing. The code snippet in Appendix A.1 demonstrates that a relatively small number of lines of C++ code defines the interface between C++ and Python, even for objects like C++ classes that are inherited from abstract base classes; wrapping objects like these can be rather challenging in general. The code in the interface file is compiled into a Python module (a shared library) and can be readily integrated into other Python code. In principle, abstract base classes could also be derived/extended at the Python level, but the C++/Python calling overhead is non-negligible. The key point here is speed, and keeping all the calculations at the C++ level reduces the interface overhead.

5. Levenberg-Marquardt

The Levenberg [1] and Marquardt [2] algorithm for the determination of model parameters through minimization of the least-squares error is one of the most well-studied and popular numerical algorithms in the optimization canon; Marquardt's seminal work [2] has been cited more than 25 thousand times as of publication. This algorithm has found such a wide range of applicability thanks to its exceptional stability and simple form. A C++ implementation of the algorithm requires only a handful of lines of code (as will be shown below), assuming a linear algebra library is already available.

The Levenberg-Marquardt implementation described in this work follows the comprehensive and clear treatment of Madsen et al. [3]. Madsen et al. modify the algorithm proposed by Levenberg and Marquardt in order to increase the stability of the method. A flowchart of the method is shown in Fig. 3.

5.1 Initialization

The Levenberg-Marquardt algorithm requires a starting point $\vec{c}_0$ from which the optimization is run. This point must be provided by the user. Once the initial guess values are loaded, some internal variables are initialized:

- Set the penalty parameter ν to 2

- The damping factor μ is set to $\mu = \tau_0 \cdot \max(\text{diag}(\mathbf{J}^T\mathbf{J}))$, where τ_0 is a user-adjustable parameter to control the initial damping. Madsen recommends a value of τ_o of 10^{-6} when the initial guess is believed to be a good estimate of the local minimum, or a much larger value (1 or more) if the initial estimate is not believed to be a good estimate of the local minimum
- The iteration counter k is set to zero

5.2 Iteration

1. The Jacobian matrix $\mathbf{J}$ and the residual vector $\vec{r}$ are evaluated (ideally, in parallel).
2. The step h_{lm} is obtained from $(\mathbf{J}^T(\vec{c})\mathbf{J}(\vec{c}) + \mu\mathbf{I})h_{\text{lm}} = -\mathbf{J}^T(\vec{c})\vec{f}(\vec{c})$
3. The new coefficients $\vec{c}_{\text{new}} = \vec{c}_{\text{old}} + \vec{h}_{\text{lm}}$ are calculated
4. The gain ratio ρ is obtained from
$$\rho = \frac{1}{2}\frac{F(\vec{c}) - F(\vec{c}_{\text{new}})}{L(0) - L(\vec{h}_{\text{lm}})} \tag{5}$$
where
$$L(0) - L(\vec{h}_{\text{lm}}) = \frac{1}{2}h_{\text{lm}}^T(\mu h_{\text{lm}} - \mathbf{J}^T\vec{f}) \tag{6}$$
and the sum of squares function F is obtained from Eq. 2.
5. If ρ is greater than zero, the step is acceptable, the damping parameter is set to $\mu = \mu \cdot \max(\frac{1}{3}, 1-(2\rho-1)^3)$ and the penalty parameter ν is reset to 2. Otherwise, if ρ is less than zero, the step is unacceptable, the damping parameter is multiplied by a factor of ν, the penalty factor ν is multiplied by 2, and the iteration is re-tried, taking a more conservative step.
6. When the error is sufficiently small, or the maximum number of iterations has been reached, stop.

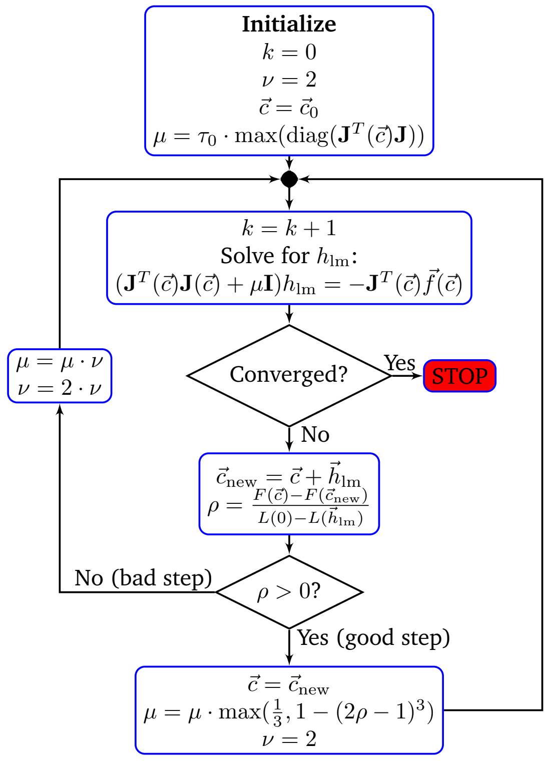

Initialize
$k = 0$
$\nu = 2$
$\vec{c} = \vec{c}_0$
$\mu = \tau_0 \cdot \max(\text{diag}(\mathbf{J}^T(\vec{c})\mathbf{J}))$
$k = k + 1$
Solve for h_{lm}:
$(\mathbf{J}^T(\vec{c})\mathbf{J}(\vec{c}) + \mu\mathbf{I})h_{\text{lm}} = -\mathbf{J}^T(\vec{c})\vec{f}(\vec{c})$
$\mu = \mu \cdot \nu$
$\nu = 2 \cdot \nu$
Converged?
Yes
STOP
No
$\vec{c}_{\text{new}} = \vec{c} + \vec{h}_{\text{lm}}$
$\rho = \frac{F(\vec{c}) - F(\vec{c}_{\text{new}})}{L(0) - L(\vec{h}_{\text{lm}})}$
No (bad step)
$\rho > 0$?
Yes (good step)
$\vec{c} = \vec{c}_{\text{new}}$
$\mu = \mu \cdot \max(\frac{1}{3}, 1 - (2\rho - 1)^3)$
$\nu = 2$

Fig. 3. Flowchart of the Levenberg-Marquardt algorithm (adapted from Algorithm 3.16 of Madsen et al. [3]) .

6. Results

In this section, we fit a decaying exponential function to artificially generated data (see Fig. 1), and by increasing the computational work per model evaluation, demonstrate near-theoretical speedup.

Here, the model function is given by

$$y_i = \exp(-c_0 x_i)\cos(c_1 x_i)\sin(c_2 x_i). \tag{7}$$

with entries in the Jacobian matrix given by

$$\left(\frac{\partial y_i}{\partial c_0}\right)_{c_{j\neq 0}} = -x_i \exp(-c_0 x_i)\cos(c_1 x_i)\sin(c_2 x_i) \tag{8}$$

$$\left(\frac{\partial y_i}{\partial c_1}\right)_{c_{j\neq 1}} = -x_i \exp(-c_0 x_i)\sin(c_1 x_i)\sin(c_2 x_i) \tag{9}$$

$$\left(\frac{\partial y_i}{\partial c_2}\right)_{c_{j\neq 2}} = x_i \exp(-c_0 x_i)\cos(c_1 x_i)\cos(c_2 x_i) \tag{10}$$

In order to be able to analyze the impact on the speedup as the computational effort per model evaluation increases, rather than using the native transcendental functions implemented in the standard math library, their Taylor series expansions are used [4]:

$$\exp x = \sum_{m=0}^{\infty} \frac{x^m}{m!} \tag{11}$$

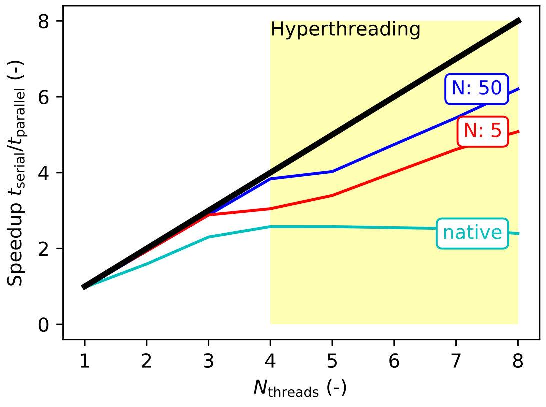
$$\sin x = \sum_{m=0}^{\infty} \frac{(-1)^m x^{2m+1}}{(2m+1)!} \tag{12}$$

$$\cos x = \sum_{m=0}^{\infty} \frac{(-1)^m x^{2m}}{(2m)!} \tag{13}$$

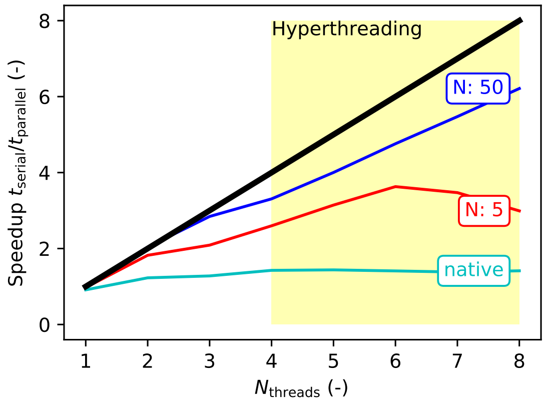
We then truncate each infinite series after the N-th term. A naïve treatment of the factorial function is used, further increasing the work per model evaluation. As a result, the computational effort per model evaluation can be extremely high for relatively small N. No effort was made to optimize the efficiency of these implementations, the point being to yield a model that serves a proxy for a practical problem without requiring additional explanation of the underlying physics.

We carried out the benchmark tests on two different machines, a single-processor desktop machine and a two-processor computer well-suited to more involved numerical computation. The results for each computer, as well as some additional discussion, are presented in the following sections.

6.1 Desktop



(a) Speedup for parallel evaluation of the residuals.



(b) Speedup for the complete Levenberg-Marquardt evaluation.

Fig. 4. Speedup for the decaying exponential fitting problem as a function of the number of threads N_{threads} and the computational effort (the number of terms N in the series expansions for the transcendental functions). The time for each evaluation is based upon the minimum of the times for a few hundred evaluations at the C++ level.

The coefficients $\vec{c} = [0.2, 3, 1.3]$ were selected for the model, and 1200 "experimental data" points were generated, linearly spaced for x in the domain [0, 2]. This optimization problem is rather sensitive to its initial value, and [0.5, 2, 0.8] is selected as the initial values for the coefficients $\vec{c}_0$. The `pybind11` wrapper was used to call the low-level C++ timing routines, and the Python code used to call the `pybind11` wrapper and generate the figures can be found in Appendix A.6 and A.7.

Figure 4a shows the results of the speedup for the evaluation of the residuals and Jacobian matrix for this model fitting exercise. As long as no hyper-threading is used, as N is increased (for up to the first four threads), the speedup approaches linear speedup. Once the number of threads exceeds the number of cores, the linear speedup is no longer maintained due to the transition to hyperthreading. When the native math functions are used, the speedup is not very significant.

Figure 4b presents the computational results of the full Levenberg-Marquardt evaluation for the decaying exponential problem. The reader should note a striking similarity in the shape of the speedup curve in Figs. 4b and 4a (particularly for large values of N). This is because for this problem, as N increases, the overall computational expense becomes dominated by the model evaluation (as opposed to the Levenberg-Marquardt iterations), and thread-parallelism can be particularly beneficial. As a result, the speedup is dominated by the speedup associated with the model evaluation.

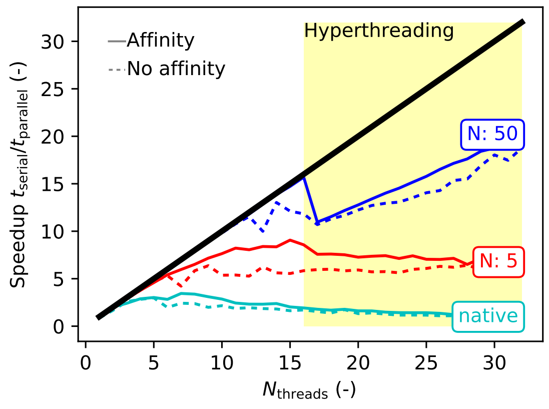
Hardware:

- Intel i7-3770 CPU @ 3.4 GHz with 4 physical cores and up to 8 threads in total (2 threads per physical core with HyperThreading technology)
- 16 GB RAM
- Windows 7 64-bit operating system
- Microsoft Visual Studio 2015 compiler

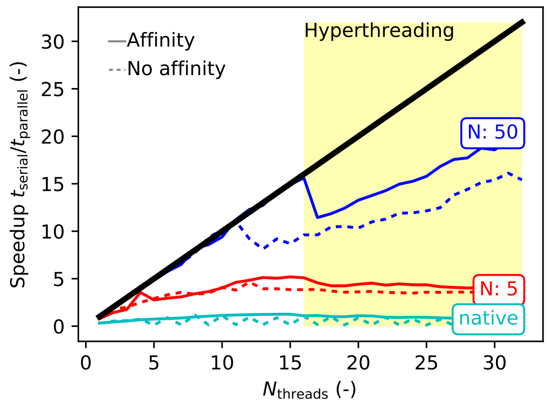
6.2 Multi-processor Machine

While the benchmark desktop computer described in Section 6.1 is a reasonable platform for general calculations, the timing tests were also run on a machine with two processors (2 $\times$ Intel Xeon E5-2667 v4 @ 3.20GHz), each with 8 physical cores and 2 hyperthreads per core, making 32 possible threads in total. The benchmark tests were run on this multi-processor machine; the results of these tests are shown in Fig. 5 for the decaying exponential optimization problem. This processor architecture is able to maintain a nearly linear speedup as the number of threads is increased for large values of N, so long as the number of physical cores is not exceeded. Beyond the number of physical cores, hyperthreading takes over, and the results with hyperthreading on this machine are significantly slower than linear speedup.

One additional complication for the multi-processor machine is that the thread affinity proved to be an important part of maximizing the the computational efficiency. Thread affinity is used to pin a thread to a given computational core. The default thread distribution scheme is architecture specific, but generally, when a thread is created, it must be attached to a core; the operating system decides what core it should be attached to given the current load on the machine. In general, the thread distribution scheme works quite well, but in the case of the evaluations of `NISTfit`, it was found that explicitly pinning each thread to a computational core yielded non-negligible improvements in computational efficiency. This speedup was achieved by first adding one thread per physical core, and then filling in with hyperthreads on the physical cores; this made it possible to maintain the linear speedup for sufficiently computationally expensive model evaluations. On the contrary, when the thread affinity is *not* set, there is a marked fall-off from linear speedup as more threads are added but while still using only the physical cores.



(a) Speedup for parallel evaluation of the residuals.



(b) Speedup for the complete Levenberg-Marquardt evaluation.

Fig. 5. Speedup for the decaying exponential fitting problem as a function of the number of threads $N_{threads}$ and the computational effort (the number of terms N in the series expansions), run on a two-processor machine with a total of 16 physical cores. The time for each evaluation is based upon the minimum of the times for a few hundred evaluations at the C++ level.

Hardware:

- Two Intel Xeon E5-2667 v4 @ 3.20GHz, each processor has 8 physical cores (and 2 threads per core with HyperThreading), for a total of 32 threads possible
- 64 GB RAM
- Xubuntu 17.10 64-bit operating system[7]
- g++ 7.2.0 compiler

7. Conclusions

This report demonstrates the use of `NISTfit` for a simplified model fitting problem. As is to be expected, as the work per model evaluation increases, the speedup from the use of threading also increases. For sufficiently expensive model evaluations, the speedup due to thread parallelism is significant, and approaches the theoretical limit of parallelism with and without hyperthreading. On the other hand, simple models do not benefit from parallelism.

The architecture of `NISTfit` is extremely flexible, and has already been applied to other practical problems like fitting model parameters for cubic equations of state. Further work is ongoing to extend the fitting paradigm to even more complex models. At its core, `NISTfit` is simply a thread-parallel evaluation tool, and it is hoped that it can be extended to use other computational engines like graphical processing units. Future work will involve the low-level coupling of the `AbstractEvaluator` architecture with additional optimization algorithms like differential evolution or evolutionary optimization.

[7]When this computer was initially configured, Xubuntu 17.04 had been installed, but it was not possible to achieve linear speedup for an unknown reason, perhaps there was a bug in the provided version of the `intel-microcode` package. Upgrading to Xubuntu 17.10 and installing the `intel-microcode` package restored the speedup of the processor.

Supplemental Materials

- Appendix A
  - A.1 src/pybind11 interface.cxx
  - A.2 include/NISTfit/abc.h
  - A.3 include/NISTfit/examples.h
  - A.4 include/NISTfit/optimizers.h
  - A.5 src/optimizers.cpp
  - A.6 evaluators.py
  - A.7 time NISTfit.py
- zip file of the code as of publication.
- doxygen-generated documentation for `NISTfit` in HTML and PDF formats.

Acknowledgments

The authors thank: Andy Prowl, whose description of the use of C++11 threading[8] was instrumental to the first implementation of threading utilized in `NISTfit`, Tobias Löw (of Steag Energy Services GmbH) for the recommendation to investigate thread affinity, Chris Muzny (of NIST) for the recommendation to add numerical Jacobian construction and Tim Blattner, Walid Keyrouz, and Derek Juba (of NIST), for many helpful recommendations with regards to profiling and optimization of C++ code.

8. References

[1] Levenberg K (1944) A method for the solution of certain non-linear problems in least squares. *Quarterly of Applied Mathematics* 2(2):164–168. http://www.jstor.org/stable/43633451.

[2] Marquardt DW (1963) An algorithm for least-squares estimation of nonlinear parameters. *Journal of the Society for Industrial and Applied Mathematics* 11(2):431–441. https://doi.org/10.1137/0111030.

[3] Madsen K, Nielsen HB, Tingleff O (2004) Methods for non-linear least squares problems (2nd ed.) http://orbit.dtu.dk/files/2721358/imm3215.pdf.

[4] Kreyszig E (2006) *Advanced Engineering Mathematics* (John Wiley and Sons), 9th Ed.

About the authors: *Ian Bell is a mechanical engineer in the Applied Chemicals and Materials Division of the Material Measurement Laboratory of NIST. He conducts research in the modeling of the thermophysical properties of pure fluids and mixtures. Matthias Kunick is a mechanical engineer in the Thermodynamics Dept. at the Zittau/Görlitz University of Applied Sciences. His current projects focus on fast property calculations for pure fluids and mixtures. The National Institute of Standards and Technology is an agency of the U.S. Department of Commerce.*

[8] http://stackoverflow.com/a/15257055/1360263