# High Performance Computing for Solving Fractional Differential Equations with Applications

Wei Zhang

Thesis submitted for the degree of Philosophiae Doctor

April 24, 2014

*Series of dissertations submitted to the*
*Faculty of Mathematics and Natural Sciences, University of Oslo*
*No. 1538*

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika Publishing.
The thesis is produced by Akademika Publishing merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

## Preface

This thesis has been submitted to the Faculty of Mathematics and Natural Sciences at the University of Oslo in fulfillment of the requirements for the degree *philosophiae Doctor (Ph.D.)*. I started my PhD research in the spring of 2011 with the Digital Signal Processing and Image Analysis (DSB) group at the Department of Informatics, University of Oslo. This work has been supervised by Professor Xing Cai, Professor Sverre Holm and Professor Wen Chen. My PhD study is financed by China Scholarship Council.

## Acknowledgements

During my PhD work, I obtained great help from many people. It's a good opportunity for me to express my sincere gratitude to them.

My work was under the supervision of Professor Xing Cai, Professor Sverre Holm and Professor Wen Chen. I'm grateful to Professor Xing Cai and Professor Sverre Holm for valuable discussions and comments. You have been constant sources of motivation, and constructive feedback. I also acknowledge the help from Professor Wen Chen, who gave me important advice and encouragement during my research.

I also express my gratitude to Dr. Peter Näsholm for his suggestions when I prepared my paper writing. Special thanks go to Dr. Wenjie Wei for his arrangement when I arrived in Norway and valuable advice during my research.

I would also thank my colleagues at the DSB group for providing a pleasant working environment. My family and friends also constantly gave me encouragement. Most of all, I thank my wife Lei for her patience, support, and understanding. Her love is the source of power for me to overcome difficulties.

# Abstract

Fractional calculus is the generalization of integer-order calculus to rational order. This subject has at least three hundred years of history. However, it was traditionally regarded as a pure mathematical field and lacked real world applications for a very long time.

In recent decades, fractional calculus has re-attracted the attention of scientists and engineers. For example, many researchers have found that fractional calculus is a useful tool for describing hereditary materials and processes. It has been used to model the properties of viscoelastic materials and anomalous diffusion. Other applications of fractional calculus include signal processing, control of dynamic system, fractal theory, finance.

In this thesis, we have investigated several applications of fractional calculus and the use of multi-core hardware architecture for solving fractional differential equations. Within heat theory, we have studied fractional generalized Cattaneo equations and pointed out that the fractional heat equations may give negative absolute temperatures. Related to elastography, we have investigated the use of a fractional wave equation to describe the shear propagation induced by radiation force. We have concluded that there is a possibility of biased estimation of shear modulus. Numerical simulation of fractional partial differential equations is a time-consuming task due to the non-local property of fractional derivatives. We have shown that optimization techniques and parallel computing can reduce the long simulation time. We have also developed performance models which can give deep understanding of the optimization techniques and predict the simulation time of both serial and parallel implementations. Last but not least, we have demonstrated that parallel solvers of three-dimensional time-fractional diffusion problems are well suited for cutting-edge parallel hardware.

## List of Publications

**I** W. Zhang and X. Cai, "Efficient implementations of the Adams-Bashforth-Moulton method for solving fractional differential equations", *Proceedings of the 5th International Federation of Automatic Control Symposium on Fractional Differentiation and its Applications, Nanjing, 2012.*

**II** W. Zhang, W. Wei and X. Cai, "Performance modeling of serial and parallel implementations of the fractional Adams-Bashforth-Moulton method", *Fractional Calculus and Applied Analysis*, Volume 17, Number 3, 2014 (to appear in June, 2014).

**III** W. Zhang, X. Cai and S. Holm, "Time-fractional heat equations and negative absolute temperatures", *Computers and Mathematics with Applications*, Volume 67, Number 1, Pages 164-171, 2014.

**IV** W. Zhang and S. Holm, "Estimation of shear modulus in media using the fractional Kelvin-Voigt model", submitted to *Computers and Mathematics with Applications*.

**V** W. Zhang and X. Cai, "Solving 3D time-fractional diffusion equations by high-performance parallel computing", submitted to journal for publication.

# Contents

# Chapter 1

# Introduction

In this chapter, I first give a brief introduction to fractional calculus. Then the basic theories of heat conduction and elastography are provided. Thereafter, I introduce software code optimization techniques that are related to solving fractional differential equations. Finally, performance models are presented to allow quantitative analysis and deeper understanding of these optimization techniques.

## 1.1 An overview of fractional calculus

Fractional calculus has at least three hundred years of history and dates back to Leibniz's letter to L'Hospital in 1695, when L'Hospital asked Leibniz, the inventor of the notation $d^n y/dx^n$, "What if n is $1/2$?". Leibniz replied: "This is an apparent paradox from which, one day, useful consequences will be drawn.".

The rather non-systematic development of fractional calculus started in the 18th century. Abel was the first researcher who used fractional calculus to solve the tautochrone problem in 1823 [5]. Liouville gave definitions of fractional derivative, and attempted to solve differential equations with fractional operators [39]. Other scientists made contributions to the development of fractional calculus including Fourier, Laplace, Riemann, Grünwald, Letnikov, Heaviside, and more. Details about the development of fractional calculus in the 18th and 19th centuries can be found in [39].

From 1900 to 1970, there was a modest development of fractional calculus. The pioneers included H. Weyl, H. Hardy, M. Riesz, H. Kober, just to name a few.

The intensive development of fractional calculus began in 1974. The first book which gave a systematic introduction to fractional calculus was written by Oldham and Spanier [42]. The first conference on fractional calculus took place at University of New Haven in 1974. The second and third international conferences were held in 1984 at University of Strathclyde and in 1989 at Nihon University [39]. Later, there have been several works done by Bagley and Torvik [54], Samko et al. [48], Podlubny [43], Miller and Ross [39], Gorenflo and Mainardi [35], and so on.

Nowadays, there are many books and journals about the mathematical analysis and applications of fractional calculus. A specialized international journal for fractional calculus, named *Fractional Calculus and Applied Analysis*, was established in 1998. An international conference series "Fractional Differentiation and its Applications" is held every second year.

There are many successful examples of applying fractional calculus in science and engineering. Viscoelasticity is one of the extensive application areas because fractional calculus is an excellent instrument for modeling materials with hereditary properties. Zhang et al. [60] used experiments on prostate to show that the stress relaxation results could be fitted by the fractional Kelvin-Voigt model. Caputo [7] presented that the stress-strain relations based on the fractional derivative is suitable for modeling wave attenuation in the anelastic media. Recently, Bentil and Dupaix [6] used the fractional Zener constitutive model to describe the mechanical behavior of swine neural tissue.

Another extensive application area of fractional calculus is anomalous diffusive and thermal transport phenomena. Metzler and Klafter [38] demonstrated that fractional equation was a complementary tool in the description of anomalous transport processes. Compte and Metzler [11] showed that some fractional Cattaneo equations were able to reproduce features which could be derived from the continuous time random walk (CTRW) model. Sun et al. [52] used a variable-index fractional model to describe transient dispersion in heterogeneous media. Luchko and Punzi [33] modelled anomalous heat transport in geothermal reservoirs via fractional diffusion equations. Ghazizadeh et al. [19] demonstrated that a fractional single phase-lag model could predict the same temperature distribution as the linear dual-phase-lag model in a non-homogeneous medium.

Fractional calculus has also attracted attention in signal processing, control of dynamic system, finance and economy, fractal theory, and so on. More application examples of fractional calculus can be found in [42, 43, 34, 5, 24, 48, 39].

There is also a systematic development in the mathematical theory of fractional calculus. The existence and uniqueness of solutions to fractional differential equation were discussed in [28, 43]. Analytical methods for solving fractional order equations, such as the Mellin transform method, power series method, method of orthogonal polynomials, homotopy perturbation method, Adomian decomposition method were discussed in [43, 55, 40]. Numerical approximation of fractional differential equations is an intensive topic currently. Finite difference [37], finite element [18], and spectral finite element method [8] are used to solve various problems. There are fruitful achievements in the theory of convergence rate, stability analysis, and error estimation. A brief overview of the methods for solving fractional differential equations can be found in this chapter later.

## 1.2 Challenges of fractional calculus

Although we have witnessed progress in both theory and application of fractional calculus, there are still many challenges. Here, I list three important challenges.

The first challenge is the definition of fractional calculus. The fractional differentiation and integration are nonlocal operators based on an integral with a singular kernel. There are several definitions of the fractional derivative, such as Caputo, Riemann Liouville, Grünwald Letnikov, Erdélyi-Kober, Hadamard, Riesz definitions. Some of these definitions are equivalent to each other under some conditions [29].

The second challenge is the physical interpretation of fractional calculus. The integer order derivatives have a clear physical and/or geometric interpretation, so their applications are easily understood. The fractional derivative is a generalization of integer order derivative and loses some of the classical properties. It is difficult to give a convincing interpretation of fractional models. For example, many fractional models are used to describe acoustic attenuation in biological media with power law characteristic. The fractional order characterizes the viscoelastic property. However, the value of the fractional order is usually decided empirically. There are discrepancies of data even for similar tissues. A similar difficulty is found in heat and diffusion theories, where the fractional order characterizes the thermal propagation and diffusion speed.

The third challenge is the time-consuming numerical solution of fractional differential equations. Fractional derivative is a non-local operator. Compared with the approximation of an integer order derivative, which only needs a few neighbouring values, the approximation of fractional derivative needs substantially more computational effort. The reason is that we must consider all values from the starting point. So numerical simulations of fractional differential equations need a larger number of floating point operations and data flow in computer memory systems. Podlubny [43] suggested the *short memory principle* to reduce the amount of computation at the expense of lower accuracy. Ford and Simpson [17] derived the *nested mesh* approach to retain the order of convergence. Deng [12] adopted the short memory principle of fractional calculus and applied an Adams-type predictor–corrector approach. Although these algorithms can reduce the computational time to some extent, solving fractional differential equation is still a challenging task. In this thesis, we discuss how to use code optimization and parallelization on multi-core computers to ease this challenge.

## 1.3 Definition of fractional calculus

In this section, we recall some definitions of fractional derivative and integral. One should notice that *fractional calculus* is the name of the theory about integrals and derivatives of arbitrary order [43]. In other words, the order can be an arbitrary real or even complex number.

There are several definitions of fractional derivative and integral, such as Grünwald-Letnikov, Riemann-liouville, Caputuo, Marchaud, Hadamard, and more. We list only the first three definitions which are the most commonly used. Other definitions can be found in [48, 5]. Here we use $D$ and $I$ to denote fractional derivative and integral, respectively.

- Grünwald-Letnikov definition
  It is a generalization of the classical integer order derivative, which for a continuous function $u(t)$ is

$$u^{(n)}(t) = \lim_{h\to 0} \frac{1}{h^n} \sum_{r=0}^{n} (-1)^r \binom{n}{r} u(t-rh), \tag{1.1}$$

  where $\binom{n}{r}$ is the binomial coefficients. If $n$ is replaced by $\alpha \in \mathbb{R}$, we get

$${}^{G}D^{\alpha}_{a,t}u(t) = \lim_{h\to 0+} \frac{1}{h^\alpha} \sum_{r=0}^{\lfloor \frac{t-a}{h} \rfloor} (-1)^r \binom{\alpha}{r} u(t-rh), \tag{1.2}$$

  where $\lfloor\cdot\rfloor$ denotes the floor function. The subscript $a$ denotes the starting point of interval.

The Grünwald-Letnikov integral of arbitrary order is

$$^{G}I_{a,t}^{\alpha}u(t) = \lim_{h\to 0+}\frac{1}{h^{\alpha}}\sum_{r=0}^{\lfloor\frac{t-a}{h}\rfloor}(-1)^{r}\binom{-\alpha}{r}u(t-rh). \tag{1.3}$$

- Riemann-Liouville definition
  The $\alpha$th order Riemann-Liouville derivative of function is

$$^{RL}D_{a,t}^{\alpha}u(t) = \frac{1}{\Gamma(n-\alpha)}\frac{d^{n}}{dt^{n}}\int_{a}^{t}\frac{u(\tau)d\tau}{(t-\tau)^{\alpha-n+1}}, \qquad n-1<\alpha<n, \tag{1.4}$$

  and the integral

$$^{RL}I_{a,t}^{\alpha}u(t) = \frac{1}{\Gamma(\alpha)}\int_{a}^{t}\frac{u(\tau)d\tau}{(t-\tau)^{1-\alpha}}. \tag{1.5}$$

- Caputo definition
  The definition of Riemann-Liouville is important in the development of fractional derivative. However, it is difficult to incorporate with physically interpretable initial conditions. Caputo solved this problem by presenting a new definition

$$^{C}D_{a,t}^{\alpha}u(t) = \frac{1}{\Gamma(n-\alpha)}\int_{a}^{t}(t-\tau)^{n-\alpha-1}\frac{d^{n}u(\tau)}{d\tau^{n}}d\tau, \qquad n-1<\alpha<n. \tag{1.6}$$

Li and Deng [29] showed that the three definitions are equivalent to each other under certain conditions. They also showed that there is a strict requirement of $u(t)$ in the Grünwald-Letnikov definition. The Riemann–Liouville derivative has a good property of reasonable generalization of the classical derivative. The Caputo fractional derivative has a good physical interpretation of the initial value and is often used in modeling and analysis. In this thesis, we always use the Caputo definition and denote it by $D^{\alpha}$. We only consider time fractional derivatives in this thesis.

## 1.4 Fractional calculus and anomalous phenomena

In this section, we present applications of fractional derivative in non-Fourier heat conduction and elastography.

### 1.4.1 Non-Fourier heat conduction

The classical Fourier's law is widely used in many engineering applications. It is an empirical law which gives the linear relationship between heat flux and temperature gradient

$$\mathbf{q} = -k\nabla T \tag{1.7}$$

where $\mathbf{q}$ is the heat flux vector, $T$ is the temperature, and $k$ is the thermal conductivity. Fourier's law leads to a parabolic heat equation (PHE). It should be noted that Eq. (1.7) is a unifying form of Fourier's law, Fick's law and Darcy's law but only with different physical parameters.

In recent decades, experiments show that the parabolic heat equation gives inaccurate results in situations like heat conduction in ultrashort duration, or at low temperatures. These phenomena are called anomalous heat conduction. The reason is that Fourier's law implies an infinite heat propagation speed. To incorporate a finite speed, many researchers have formulated modifications of Fourier's law. For example, Cattaneo [9] modified Fourier's law by adding a relaxation term to Eq. (1.7)

$$\mathbf{q} + \tau \frac{\partial \mathbf{q}}{\partial t} = -k \nabla T, \tag{1.8}$$

and obtained the hyperbolic heat equation (HHE)

$$\frac{1}{c^2}\frac{\partial^2 T}{\partial t^2} + \frac{1}{a}\frac{\partial T}{\partial t} = \nabla^2 T, \tag{1.9}$$

where $a$ is the thermal diffusivity, $c = \sqrt{a/\tau}$ the thermal propagation speed.

In recent decades, some researchers have introduced non-local models to describe anomalous phenomena. These models generalize classical laws. For example, Fourier's law can be generalized as [22]

$$\mathbf{q}(t) = -k \int_0^\infty K(u) \nabla T(t-u) du. \tag{1.10}$$

Letting $t - u = \tau$ and choosing 0 as a staring point, we have

$$\mathbf{q}(t) = -k \int_0^t K(t-\tau) \nabla T(\tau) d\tau. \tag{1.11}$$

Fourier's law is a special case of Eq. (1.11) if we choose the kernel $K(t)$ as the Dirac delta function. If we choose a "long-tail" power law kernel, such as

$$K(t-\tau) = \frac{\alpha}{\Gamma(1-\alpha)}(t-\tau)^{-\alpha-1}, \quad 0 < \alpha < 1, \tag{1.12}$$

then the flux can be interpreted in terms of a fractional integral

$$\mathbf{q} = -k \cdot I^{1-\alpha} \nabla T, \tag{1.13}$$

and one gets the time fractional heat equation (FHE) with order $\alpha$

$$D^\alpha u(t) = a \Delta T. \tag{1.14}$$

There is a wide range of choice for $\alpha$, and different values of the fractional order $\alpha$ in Eq. (1.14) give different heat conduction property. In diffusion theory, it is called subdiffusion for $0 < \alpha < 1$, and superdiffusion for $1 < \alpha < 2$. As one of the aforementioned challenges, the value of the fractional order is empirical and lacks clear physical interpretation. Figure. 1.1 shows the numerical solutions of FHE with different values of $\alpha$. It is a one dimensional problem with the initial temperature $T = 1$ and boundary $T = 0.2$. The orders $\alpha = 0.6$, 1, and 1.6 are chosen for experiments. We can see that for $\alpha = 1.6$, the equation is more like a wave equation and there is a time delay for the heat to reach the center. For $\alpha = 0.6$ and 1, there is

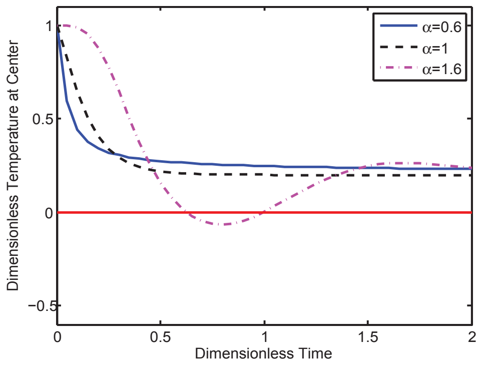


**Figure 1.1:** *The numerical solution of time fractional heat equation (FHE) varies with the fractional order $\alpha$=0.6, 1, 1.6. The thermal diffusivity $a = 1$.*

no such delay which implies an infinite heat propagation speed.

Although the hyperbolic heat equation and fractional heat equation can give finite heat propagation speed, they may introduce other problems. Rubin [47] proved that Cattaneo's equation may violate the second law of thermodynamics. Bai and Lavine [4] showed that HHE can give negative temperatures, see Figure 1.2. For time equals to 0.6, the hyperbolic equation gives negative temperature. The fractional heat equation also has the same problem, see Fig. 1.1. Details about the problem are discussed in paper III.

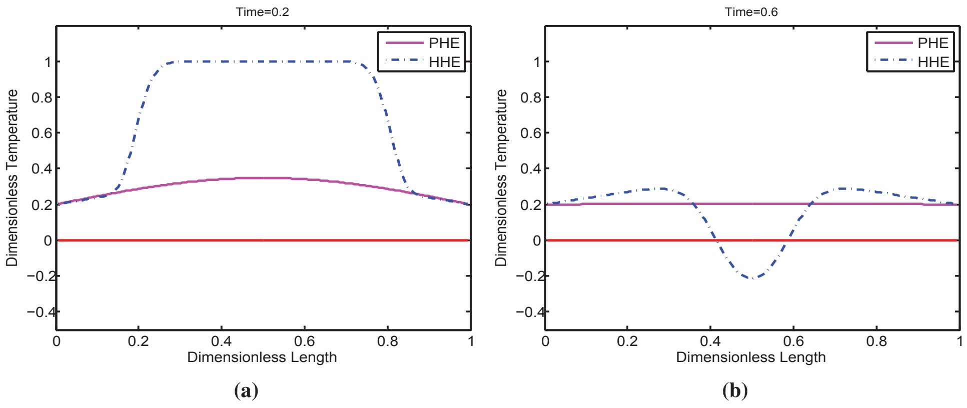


**Figure 1.2:** *Temperature distribution for $T = 0.2$ and $T = 0.6$. The hyperbolic heat equation (1.9) gives negative temperatures.*

## 1.4.2 Modeling arbitrary power law attenuation

One extensive application area of fractional calculus is viscoelasticity. The relationship between stress $\sigma$ and strain $\epsilon$ for solids is

$$\sigma(t) = E\epsilon(t), \tag{1.15}$$

and for Newtonian fluids

$$\sigma(t) = \eta \frac{d\epsilon(t)}{dt}. \tag{1.16}$$

The mechanical representation of Hooke's elastic element and Newton's viscous element are spring and dashpot, respectively [43], see Figure 1.3 (a) and (b). These two elements can be combined in different ways to capture both elastic and viscous properties. For example, the Maxwell model has the following relation

$$\frac{d\epsilon}{dt} = \frac{1}{E}\frac{d\sigma}{dt} + \frac{\sigma}{\eta}. \tag{1.17}$$

Another example is the Voigt model

$$\sigma = E\epsilon + \eta \frac{d\epsilon}{dt}. \tag{1.18}$$

Mechanical representation of the Maxwell and Voigt models are depicted in Figure 1.3 (c) and (d).

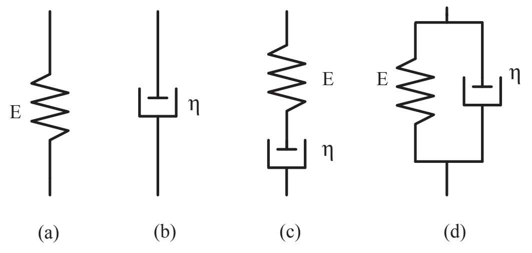


**Figure 1.3:** *Mechanical models of spring (a), dashpot (b), Maxwell's model (c) and Voigt's model (d).*

There are some disadvantages of both Maxwell's and Voigt's models. It is well known that wave attenuation in many materials, such as human tissue, polymers, obeys a power law

$$a(\omega) = a_0\omega^y, \tag{1.19}$$

where $\alpha_0$ is the absorption coefficient in Np(rad/s)$^{-y}$m$^{-1}$, $\omega$ the angular frequency in rad/s, and $y$ the power law exponent. In medical ultrasound, human tissue has $y$ ranging from 1 to 1.7. For fluid, such as distilled water and certain oils, we have $1 < y < 2$ [53].

Sinkus et al. [49] have shown that the Voigt model is in contradiction with power law since it predicts a constant dynamic modulus and linearly rising loss modulus. The same problem also exists in the Maxwell model. Furthermore, partial differential wave equations based on the classical stress-strain relationship lead to an integer-order equation which can only predict attenuation with $y = 0$ and 2. Attenuation with $y$ different from 0 and 2 is called anomalous attenuation.

Fractional calculus is a powerful tool in describing anomalous attenuation with fewer parameters. From Eq. (1.15) and (1.16), we can see the stress is in proportion to the zeroth and first derivatives of strain for solids and liquids. It is natural to say that viscoelastic materials are

**Table 1.1:** *Asymptotic dispersion relation for the Caputo equation*

| | $\omega\tau \ll 1$ | $\omega\tau \gg 1$ |
|---|---|---|
| $a(\omega) \propto$ | $\omega^{\alpha+1}$ | $\omega^{1-\alpha/2}$ |
| $c_p \propto$ | $\omega^{\alpha}$ | $\omega^{\alpha/2}$ |

"intermediate" ones which have the following stress-strain relation [43]

$$\sigma(t) = E_0 D^{\alpha}\epsilon(t). \tag{1.20}$$

The fractional Kelvin-Voigt stress and strain relation is the combination of such "intermediate" materials and solids

$$\sigma(t) = E\left[\epsilon(t) + \tau_\sigma^\alpha D^\alpha \epsilon(t)\right]. \tag{1.21}$$

To derive a wave equation, we also need the strain-displacement relation

$$\epsilon(t) - \frac{\partial u}{\partial x} = 0, \tag{1.22}$$

and the principle of conservation of momentum

$$\frac{\partial}{\partial x}\sigma(t) - \rho\frac{\partial^2 u(x,t)}{\partial t^2} = 0. \tag{1.23}$$

Combing Eq. (1.21), (1.22) and (1.23), we get the Captuo equation

$$\nabla^2 u - \frac{1}{c_0^2}\frac{\partial^2 u}{\partial t^2} + \tau_\sigma^\alpha D^\alpha(\nabla^2 u) = 0. \tag{1.24}$$

Holm and Sinkus [27] analyzed the dispersion relation of Eq. (1.24) by assuming a plane wave solution $u(x,t) = \exp\{i(\omega t - kx)\}$. The dispersion relation is divided into two regimes. For low product $\omega\tau$, the exponent $y$ and fractional order has the relation $y = \alpha + 1$. It covers most of compressional wave cases. For high product $\omega\tau$, $y = 1 - \alpha/2$, it covers shear wave cases. The asymptotes of frequency-dependent absorption and phase velocity can be found in Table 1.1.

There are a set of wave equations with different fractional loss operators. The fractional Zener model is another generalization of stress and strain relation [26]

$$\sigma(t) + \tau_\epsilon^\beta D^\beta \sigma(t) = E\left[\epsilon(t) + \tau_\sigma^\alpha D^\alpha \epsilon(t)\right], \tag{1.25}$$

which leads to the fractional Zener wave equation

$$\nabla^2 u - \frac{1}{c_0^2}\frac{\partial^2 u}{\partial t^2} + \tau_\sigma^\alpha D^\alpha(\nabla^2 u) - \frac{\tau_\epsilon^\beta}{c_0^2} D^{\beta+2} u = 0, \tag{1.26}$$

where $\tau_\sigma$ is the creep time. Chen and Holm proposed in [10] an equation with the fractional

Laplacian operator

$$\frac{1}{c_0}\frac{\partial^2}{\partial t^2} = \nabla^2 p + \tau\frac{\partial}{\partial t}(-\nabla^2)^{\alpha/2}p, \tag{1.27}$$

where $p$ is the pressure.

It should be noted there are other approaches which can model power law attenuation. Szabo [53] presented a casual time domain wave equation which complies well with arbitrary power law frequency. Wismer and Ludwig [58] proposed a frequency domain model via the Laplace transform which is only applicable for linear cases. The multiple relaxation model [41] has a physically based description of acoustic loss mechanism and is considered adequate for relaxation-dominated attenuation. But it needs extensive computational effort and estimation of many obscure parameters.

## 1.5 Solving fractional differential equations

In this section, we present a short overview of both analytical and numerical methods for solving fractional differential equations.

### 1.5.1 Analytical methods

A lot of work has been done for analytically solving fractional differential equations . For example, Gorenflo et al. [20] gave the scale-invariant solutions to the mixed problem of signalling type for the time-fractional diffusion-wave equation. The solutions are in terms of the Wright function. Liu et al. [32] obtained the complete solution to a time-fractional advection dispersion equation by using variable transformation, Mellin and Laplace transforms, and properties of H-functions. The fundamental solutions to the Cauchy problem and to the source problem as well as the associated stresses were obtained by Povstenko [45]. Gupta and Singh [21] presented the approximate analytical solutions to the time fractional nonlinear Fornberg–Whitham equation, and the explicit solutions are obtained using the homotopy perturbation method.

### 1.5.2 Numerical methods

When solving real-world problems, it is difficult to obtain the analytical solutions to fractional differential equations. Even if the analytical solution is available, it may not be convenient to use in practice. It is important to develop numerical algorithms which have sufficient accuracy, stability, and convergence.

Numerical simulation of differential equations of integer order is not a new topic and has been explored by many scientists and engineers. However, efficient and accurate numerical methods for solving fractional differential equations and their theoretical analysis are far less advanced.

Adolfsson et al. [2] developed an efficient numerical method which can integrate the constitutive response of fractional-order viscoelasticity. The method can deal with variable time steps, and the priori and posteriori error estimates of the method are proved. Diethelm et al.

[16, 14] discussed an Adams-type predictor-corrector method for an ordinary fractional differential equation. They also extended the method to multi-term equations. Lin and Liu [31] proposed a high-order approximation scheme for solving a nonlinear fractional-order ordinary differential equation. Meerschaert et al. [36] used the alternating directions implicit method to solve two-dimensional initial-boundary-value fractional partial differential equations with variable coefficients. They showed that when using the novel shifted version of finite difference approximation of the Grünwald definition, the scheme has good numerical property regarding consistency, stability, and convergence. Sousa [50] used upwind, central and Lax–Wendroff schemes to solve fractional advection–diffusion problems. The convergence rate and stability of these schemes are discussed. Podlubny et al. [44] proposed the matrix approach which unifies the numerical solutions of both fractional and integer-order differential equations. Deng [13] developed the finite element method for the numerical resolution of the space and time fractional Fokker-Planck equation. Carella [8] developed the least-squares spectral element method and demonstrated that the method is well-suited for dealing with the numerical difficulties inherent to fractional differential operators.

Most numerical methods for solving fractional differential equations are based on approximations of the fractional differential operators using appropriate formulas[5]. From the Caputo definition of fractional derivative, we can say what we are interested in is the approximation of integral operators. Here, we use two examples to show the approximation of integral operators when solving fractional ordinary and partial differential equations.

The first example is the *fractional Adams-Moulton formula* proposed by Diethelm et al. [16]. It can be used for solving fractional ordinary differential equations. The target mathematical model is an initial-value problem involving fractional derivative

$$D^\alpha y(x) = f(x, y(x)), \quad y^{(k)}(0) = y_0^{(k)}, k = 0, 1, ...\lceil\alpha\rceil - 1, \tag{1.28}$$

where $\alpha > 0$, $\lceil\cdot\rceil$ is the ceiling function. Eq. (1.28) is equivalent to the Volterra integral equation

$$y(x) = \sum_{k=0}^{\lceil\alpha\rceil-1} y_0^{(k)} \frac{x^k}{k!} + \frac{1}{\Gamma(\alpha)} \int_0^x (x-t)^{\alpha-1} f(t, y(t))dt. \tag{1.29}$$

An approximate solution can be found by using the product trapezoidal quadrature formula to replace the integral. In other words, the following approximation is applied

$$\int_0^{t_{n+1}} (t_{n+1} - z)^{\alpha-1} g(z)dz \approx \int_0^{t_{n+1}} (t_{n+1} - z)^{\alpha-1} \tilde{g}_{n+1}(z)dz, \tag{1.30}$$

where $\tilde{g}_{n+1}$ is a piecewise linear interpolation of $g$ with nodes chosen at the $t_j$. Applying the standard techniques from quadrature theory, the right-hand side of Eq. (1.30) can be written as

$$\int_0^{t_{n+1}} (t_{n+1} - z)^{\alpha-1} \tilde{g}_{n+1}(z)dz = \frac{h^\alpha}{\alpha(\alpha+1)} \sum_{j=0}^{n+1} a_{j,n+1} g(t_j), \tag{1.31}$$

where

$$a_{j,n+1} = \begin{cases} n^{\alpha+1} - (n-\alpha)(n+1)^{\alpha}, & j = 0, \\ (n-j+2)^{\alpha+1} + (n-j)^{\alpha+1} - 2(n-j+1)^{\alpha}, & 1 \le j \le n, \\ 1, & j = n+1. \end{cases} \tag{1.32}$$

Now we get the *corrector* formula. The same principle applies for the *predictor* formula. The details can be found in [16]. To summarize, the formulae for both predictor and corrector are

1. Predictor

$$y_{n+1}^{P} = \sum_{k=0}^{\lceil\alpha\rceil-1} \frac{x_{n+1}^{k}}{k!} y_0^{(k)} + h^{\alpha} \sum_{k=0}^{n} b_{n-k} f(x_k, y_k), \tag{1.33}$$

   with weights

$$b_{\mu} = \frac{(\mu+1)^{\alpha} - \mu^{\alpha}}{\Gamma(\alpha+1)}. \tag{1.34}$$

2. Corrector

$$y_{n+1} = \sum_{k=0}^{\lceil\alpha\rceil-1} \frac{x_{n+1}^{k}}{k!} y_0^{(k)} + h^{\alpha} \left( c_n f(x_0, y_0) + \sum_{k=1}^{n} a_{n-k} f(x_k, y_k) + \frac{f(x_{n+1}, y_{n+1}^{p})}{\Gamma(\alpha+2)} \right), \tag{1.35}$$

with weights $a_{\mu}$ and $c_{\mu}$

$$a_{\mu} = \frac{(\mu+2)^{\alpha+1} - 2(\mu+1)^{\alpha+1} + \mu^{\alpha+1}}{\Gamma(\alpha+1)}, c_{\mu} \frac{\mu^{\alpha+1} - (\mu-\alpha)(\mu+1)^{\alpha}}{\Gamma(\alpha+2)}. \tag{1.36}$$

When solving fractional partial differential equations, the finite difference method is often used to approximate the fractional derivative. For example, the approximation to the Caputo derivative is [30]

$$\begin{aligned} D^{\alpha}u(x, t_{n+1}) &= \frac{1}{\Gamma(1-\alpha)} \sum_{j=0}^{n} \int_{t_j}^{t_{j+1}} (t_{n+1} - \eta)^{-\alpha} \frac{\partial u(x,\eta)}{\partial \eta} d\eta \\ &= \frac{1}{\Gamma(1-\alpha)} \sum_{j=0}^{n} \int_{t_j}^{t_{j+1}} (t_{n+1} - \eta)^{-\alpha} \frac{u(x, t_{j+1}) - u(x, t_j)}{\Delta t} d\eta + r_{\Delta t}^{n+1} \\ &\approx \frac{(\Delta t)^{1-\alpha}}{\Gamma(2-\alpha)} \sum_{j=0}^{n} \left[(n-j+1)^{1-\alpha} - (n-j)^{1-\alpha}\right] [\frac{u(x, t_{j+1}) - u(x, t_j)}{\Delta t}], \end{aligned} \tag{1.37}$$

where $0 < \alpha < 1$. More details about these two approximations can be found in [16, 30].

From Eqs. (1.33), (1.35) and (1.37), we can see that to compute a new value at point $n+1$, we need all values from points 0 to $n$. This is typically implemented using loops and provides chances for parallelization. The non-local property of the fractional derivative means that numerical simulation of fractional differential equations is a time consuming task due to large amounts of floating point operations and data traffic in the memory system.

## 1.6 Optimization and parallelization

As mentioned in the previous sections, due to the non-locality of the fractional differential operator, numerical simulation of fractional partial differential equations is a computationally heavy task. A lot of floating point operations and data transfers in the memory systems are needed if we want to have accurate results or solve large problems. For most numerical methods for solving fractional differential equations, the fundamental non-local properties remain the same. How to deal with the non-locality and reduce the simulation time is an important topic. In this section, we show how to use optimizing strategies and parallelization to reduce the simulation time.

In the last two decades, we have witnessed a rapid development of microprocessors. The central processing unit (CPU) can provide $10^9$ floating-point operations per second (GFLOPS). For example, the Intel Xeon Processor E5-2670's double-precision floating point theoretical performance is 26.4 GFLOPS per core. However, such a strong floating-point capacity can not be fully utilized due to the low bandwidth of memory. The maximum memory bandwidth of Xeon E5-2670 is only 51.2 GB/s (4 memory channels for serving 8 cores). It means that the achievable performance of floating-point operations, for example, dot product, can only be 3.2 GFLOPS. The unbalanced performance regarding the CPU peak performance and memory bandwidth calls for data access optimization.

### 1.6.1 Locality and data reuse

The key aspect of optimizing memory access is the reuse of data in registers and caches. In a modern CPU, registers and caches are used to reduce the instruction and data access latencies. Registers and caches have a higher bandwidth than the main memory. The typical bandwidths of different devices in computer systems are shown in Figure 1.4.

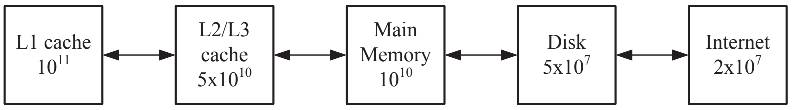


**Figure 1.4:** *Typical bandwidths in computer systems. The unit of the data in the boxes is bytes/sec. The data is obtained from [23].*

Since fetching data from a cache is much faster than from the main memory, we want to reuse data in caches as much as possible. There are two important concepts: *temporal* and *spatial* locality. Temporal locality means the data will be in cache within a short time of its last use. Spatial locality means if a particular memory location is accessed, the neighbouring locations will be referenced. These two concepts are important for optimizing codes. Here we use *loop fusion* and *loop unrolling* as examples to demonstrate how to take advantage of data locality.

Loop fusion means replacing multiple loops with a single one. Look at the following code segment

```
for (i = 0; i < N; i++)
    sum1+=a[i]*c[i];
for (i = 0; i < N; i++)
```

```
    sum2+=b[i]*c[i];
```

For each index $i$, the value $c[i]$ needs to be reloaded. The loads of $c[i]$ can be halved by combining the two loops into one

```
for (i = 0; i < N; i++){
    sum1+=a[i]*c[i];
    sum2+=b[i]*c[i];
    }
```

The performance will be enhanced greatly when the array size $N$ is large and the $c$ values are out of cache before being reloaded. One application example of this technique can be found in the implementation of fractional Adams-Bashforth-Moulton scheme. The effect of loop fusion is discussed in papers I and II.

Loop tiling or loop blocking is another important technique to explore data locality. The following code segment is used in the implementation of matrix-vector multiplication

```
for (i = 0; i < N; i++) {
    c[i] = 0;
    for (j = 0; j < N; j++) {
      c[i] = c[i] + a[i][j] * b[j];
    }
  }
```

For large $N$, the accessed array elements in each `i`-indexed loop iteration, arrays $a[i][j]$ and $b[j]$ may cross cache lines and cause performance degradation. Loop tiling can increase locality. After loop tiling using $2 \times 2$ blocks, the code becomes

```
for (i = 0; i < N; i += 2) {
    c[i] = 0;
    c[i + 1] = 0;
    for (j = 0; j < N; j += 2) {
      for (x = i; x < min(i + 2, N); x++) {
        for (y = j; y < min(j + 2, N); y++) {
          c[x] = c[x] + a[x][y] * b[y];
        }
      }
    }
  }
```

Now the *j*-indexed loop is divided into smaller blocks. For the small loops, the $a[x][y]$ and $b[y]$ values can be reused.

### 1.6.2 Vectorization

Loop vectorization is an important technique which can lead to significant performance improvement. It converts a scalar operation, which processes a single pair of operands at a time, to a vector operation which can process multiple operands at once. It is supported by Intel's SSE, AVX and AMD's 3D Now! instruction sets. For example, a CPU with the AVX instruction set could perform 4 floating point operations of double precision at a time. The following code compares a scalar operation with the vectorized counterpart

```
for (i = 0; i < 100; i++)
    c[i] = a[i]*b[i];      /* scalar  operation */

for (i = 0; i < 100; i+=4)
```

```
c[i:i+3] = a[i:i+3]*b[i:i+3];       /* vectorization */
```

Loop vectorization can also fully utilize registers by reducing superfluous data loads from the L1 cache.

We have discussed different techniques to optimize memory access. Although the performance of a program is often determined by how quickly data is moved within the entire memory system, we should also pay attention to the floating point operations in order to avoid CPU doing 'heavy task' and becoming the bottleneck. For a programmer, it is easy to implement a numerical scheme by one to one translation without considering the code performance. However, the simulation time can be reduced by changing an 'expensive' operation into a 'cheaper' one. For instance, the expensive power function $x^2$ and division $a/2$ should be substituted by $x * x$ and $0.5 * a$, respectively. It is recommended to achieve such substitutions manually before optimizing memory accesses.

It should be noted that for modern compilers, there is a collection of optimization options. Code optimizations can be achieved by compilers automatically. However, the programmer should know the basic code optimization strategies and avoid stumbling blocks for compiler-based optimization. In some cases, optimization should be done manually since the compiler has a complex job of transforming high-level programming language into machine language, there is no guarantee of obtaining high performance by only depending on the compiler.

### 1.6.3 Parallelization

The appearance of multi-core architecture has a great impact on scientific computing and has been widely adopted by all major processor manufactures. It also provides an opportunity to tackle the computational challenges of solving fractional differential equations.

Until now, there exists little work on the use of parallel computing for solving fractional differential equations. Diethelm [15] developed a parallel version of the fractional Adams-Bashforth-Moulton algorithm, and showed that the algorithm has good scalability on a shared-memory platform.

Our parallelization method is straightforward. Specifically, we convert a serial loop into a parallel one if there is no dependence between the loop iterations. For example, a *for*-loop is executed in parallel by several threads or processes.

In an OpenMP implementation, we insert a compiler directive before a loop to fork additional threads to carry out the work. Let's use dot product as example, the OpenMP implementation is

```
#pragma omp parallel for reduction (+: sum)
for (i=0; i<N; i++) {
sum+=a[i]*b[i];
}
```

In an MPI implementation, each process has its own ID and computes a section of the loop. The code segment is listed below

```
int my_start = (my_rank*N)/num_procs+1;
int my_stop = ((my_rank+1)*N)/num_procs;
for (i=my_start; i<=my_stop; i++) {
sum+=a[i]*b[i];
}
```

```
/∗MPI_Reduce is used to reduce values on all processes∗/
```

As stated in Section 1.5.2, due to the non-locality of fractional operators, to compute the value at point $N+1$, the values from the starting point to $N$ are all needed. Assuming the iteration is implemented using a *for* loop, the *for* loop can be executed in parallel for each $N$.

### Efficient parallel programming

Although multi-core systems have provided opportunities for numerical simulation of fractional differential equations, there are still some challenges. Traditional programming and software strategies must be updated in order to take advantage of the additional computing resources. Programmers must concern about the diversity of parallel platforms, portability of the program, management of independent threads or processes, concurrency and communication, load balances. Here we list three issues worthy attention when developing efficient parallel programs.

- UMA and NUMA

  First, we should make optimal use of the memory system on shared-memory platforms. At present, there are two kinds of shared-memory computers: uniform memory access (UMA) and nonuniform memory access system (NUMA). Figure 1.5 demonstrates a dual-socket quad-core UMA system. Two cores share 4MB L2 cache. All cores share the physical memory uniformly. In other words, the latency and bandwidth of memory accesses are the same for all cores. One drawback of the UMA architecture is the scalability. After the number of processors increases, CPU-to-memory connection will become the bottleneck. This leads to the appearance of the NUMA architecture. Figure 1.6 shows a dual-socket hexa-core NUMA system. Each socket has its own local memory, and the two memories are connected. NUMA systems are ubiquitous nowadays. However, data

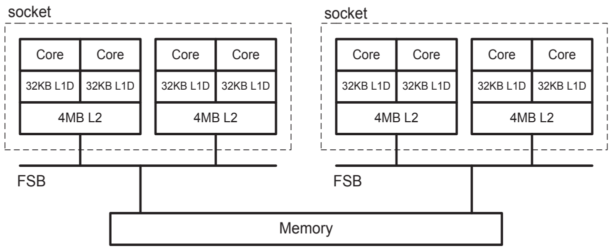


**Figure 1.5:** *A UMA system with dual-socket quad-core Clovertown X5355 CPU.*

  locality is an important issue on NUMA systems. There will be a great memory access latency if the cores have remote memory accesses. Let's use the OpenMP implementation of dot-product as an example. The code segment is

```
double a[N];
double b[N];
  for (i=0; i<N; i++){
    a[i] = 1.0∗i;
    b[i]= 1.0∗i/N;
}
```

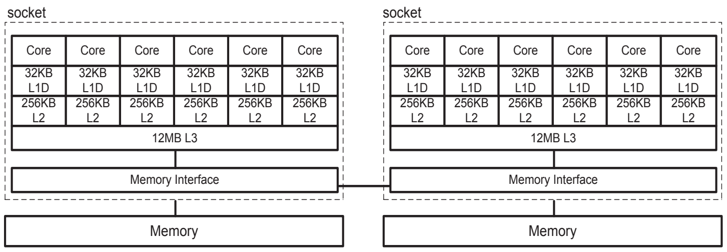


**Figure 1.6:** *A NUMA system with dual-socket hexa-core Westmere-EP L5640 CPU.*

```
#pragma omp parallel  for   reduction (+: sum)
 for  (i=0;  i<N; i++){
     sum+=a[i]*b[i ];
 }
```

This code will suffer from performance degradation since the initialization of the arrays is executed serially. All memory pages belonging to the arrays `a` and `b` will be mapped into local domain. In the parallel region, some threads will suffer from remote memory accesses. To avoid this problem, we can take advantage of the first touch policy. Specifically, initialization of the arrays is executed in parallel by all threads, so that each thread touches a portion of arrays `a` and `b`. The two arrays are thus distributed evenly among all threads. The implementation is

```
#pragma omp parallel  for
    for  (i=0;  i<N; i++){
     a[i] = 1.0*i;
     b[i]= 1.0*i/N;
}
```

- False sharing

  False sharing is another issue worthy attention. It happens when the same cache line is modified by multiple threads. The above OpenMP code for initialization of array `a` and `b` may cause false sharing if the array length $N$ is small and there is a large number of threads, or the chunk size is small.

  We can avoid false sharing by using a technique called `padding`. It means inserting suitable number of empty bytes in the array so that the data accessed by different threads resides on different cache lines.

- Processor affinity

  There is a potential performance enhancement of parallel program if we bind a process or a thread to a designated core. Spreading all processes or threads across all sockets evenly will result in a higher memory bandwidth than binding all of them to one socket. Furthermore, processor affinity can also take advantage of data locality since the data may stay in the local cache if a process or a thread is executed on the same core. Parallel programs will benefit from processor affinity if there is synchronization between adjacent processes or threads [23].

The implementation of processor affinity varies on different platforms and programs. For example, processor affinity in Open MPI v1.4.x can be achieved by adding command-line `--bind-to-core` to `mpirun`. For the gcc compiler, we can bind threads to a specific core using the `GOMP_CPU_AFFINITY` variable. All parallel programs in this thesis used processor affinity.

## 1.7 Performance modeling

The multi-core processor brings great opportunities for scientific computing. Although the microprocessors vary, they follows a similar architecture. We may care about the performance if we run the same code on different platforms. An easy-to-understand performance model can provide performance guidelines for scientific computing on different platforms. It should not only describe the amount of floating point operations, how data is transferred, abstract the characteristic of the computer system [46], but also pinpoint the bottleneck of the system and evaluate the code implementation. More importantly, it can give us a deeper understanding of our program.

There are many methods about performance modeling. Williams et al. [57] presented the roofline model which provided valuable insight into factors affecting the performance and highlighted computer system bottleneck. Adhianto and Chapman [1] proposed a novel and cost efficient approach which was based on both static analysis and feedback from a runtime benchmark. Their model can evaluate OpenMP, MPI and hybrid MPI+ OpenMP program. Wu and Taylor [59] showed a performance modeling framework which is based on memory bandwidth contention time and a parameterized communication model. Aversa et al. [3] used HeSSE, along with XML-based prototype language MetaPL to predict performance of hybrid MPI/OpenMP code.

Our model is based on the assumption that for modern computers, floating point operations and data transfer in different levels of the memory system are executed simultaneously. Compared with the roofline model in [57], which only focuses on floating point operations and memory performance, our model also investigate possible bottleneck in different cache levels and can be used for both serial and parallel programs. Specifically, assuming there are three level caches, the simulation time is determined by the maximum time usage of the following five events: floating point operations in CPU, data transferred from L1 cache to register, L2 cache to L1 cache, L3 cache to L2 cache, memory to L3 cache. More details will be listed in the following part.

In our model, the L1 cache has the largest data traffic volume since we assume that there is no data reuse in registers. In other words, all data in the registers have to be loaded from L1 cache. For the L2, L3 and main memory, data reuse is considered.

### 1.7.1 Serial performance model

Assuming there are three levels of caches in a system, the hardware can be abstracted to a group of parameters [56]

1. the peak floating point performance of a single core, denoted by $F$,

2. the bandwidths of reading data from L1 cache to register, denoted by $B^r_{L1}$, from L2 to L1 cache $B^r_{L2}$, L3 to L2 cache $B^r_{L3}$, and memory to L3 cache $B^r_M$, where the superscript $r$ means read.

Since there are more read operations than write operations when solving fractional differential equation, we neglect the write bandwidth. For a given problem size, we also have similar parameters describing the problem

1. the total number of floating point operations, denoted by $n_{flop}$,

2. the data volume needed to be read from L1 cache to registers, denoted by $n^r_{L1}$, from L2 to L1 cache $n^r_{L2}$, L3 to L2 cache $n^r_{L3}$, and memory to L3 cache $n^r_M$.

The time usage of a serial computation is

$$T_{serial} = \max(\frac{n_{flop}}{F}, \frac{n^r_{L1}}{B^r_{L1}}, \frac{n^r_{L2}}{B^r_{L2}}, \frac{n^r_{L3}}{B^r_{L3}}, \frac{n^r_M}{B^r_M}). \tag{1.38}$$

The peak floating point performance can be found in hardware specifications. The bandwidth $B^r_{L1}$, $B^r_{L2}$, $B^r_{L3}$ and $B^r_M$ can be obtained from benchmark software, such as STREAM2 [51].

We use the fractional Adams-Bashforth-Moulton scheme again to demonstrate the serial model. The following code segment is the baseline implementation

```
for (j=0; j<N; j++) {
    for (k=0; k<=j; k++)
      sum_b += b[j-k]*f(y[k]);

    for (k=1; k<=j; k++)
      sum_a += a[j-k]*f(y[k]);
}
```

There are 4 data loads: $b[j-k]$, $a[j-k]$, and $y[k]$ which is loaded twice. Assuming all data is double precision and occupies 8 bytes, the total number of floating point operations and the data transfer volume are

1. $n_{flop} = 4 \times \frac{1}{2} \times N(N+1) = 2N(N+1)$,

2. $n^r_{L1} = n^r_{L2} = n^r_{L3} = n^r_M = 4 \times \frac{1}{2}N(N+1) \times 8 = 16N(N+1)$.

Therefore, the time usage of the baseline implementation is

$$T_{baseline} = N(N+1)\max(\frac{2}{F}, \frac{16}{B^r_{L1}}, \frac{16}{B^r_{L2}}, \frac{16}{B^r_{L3}}, \frac{16}{B^r_M}). \tag{1.39}$$

### 1.7.2 Parallel performance model

The parallel model adopts the same philosophy as that of the serial model, but uses aggregated hardware parameters. Supposing there are $p$ cores, the hardware parameters for a multi-core architecture can be abstracted as

1. the peak floating point performance of $p$ cores, denoted by $pF$,

2. the aggregated bandwidths of reading data from L1 caches to registers, denoted by $B^{r,p}_{L1}$, from L2 to L1 caches $B^{r,p}_{L2}$, from L3 to L2 caches $B^{r,p}_{L3}$, and from memory to L3 caches $B^{r,p}_{M}$.

The parallel performance model is

$$T_{parallel} = \max(\frac{n_{flop}}{pF}, \frac{n^r_{L1}}{B^{r,p}_{L1}}, \frac{n^r_{L2}}{B^{r,p}_{L2}}, \frac{n^r_{L3}}{B^{r,p}_{L3}}, \frac{n^r_{M}}{B^{r,p}_{M}}). \tag{1.40}$$

The bandwidths $B^{r,p}_{L1}$, $B^{r,p}_{L2}$, $B^{r,p}_{L3}$ and $B^{r,p}_{M}$ can be obtained from the parallel version of STREAM2.

In summary, our model is easy to understand and contains a small number of parameters. It can estimate simulation time and pinpoint the performance bottleneck. Detailed information can be found in papers II and V.

# Chapter 2

# Summary of Papers

This section summarizes my publications. The details can be found in the papers.

## 2.1 Paper I

Paper I considers efficient implementations for solving a system of fractional differential equations. The Bagley-Torvik equation was chosen and rewritten as a system of four coupled scalar fractional differential equations with the fractional order $\alpha = 1/2$. We demonstrated code optimization techniques like loop fusion, loop unrolling and alternating directions for loop traversal. The ratio of floating point operation versus memory read is optimized gradually. The vectorization was left to compilers.

In the MPI and OpenMP implementations, parallelization was explored by dividing the inner `for`-loop by a number of processes or threads. Each loop segment was executed on different processors. The implementation is similar to the example codes shown in Section 1.6.3. The outer `for`-loop is still executed serially.

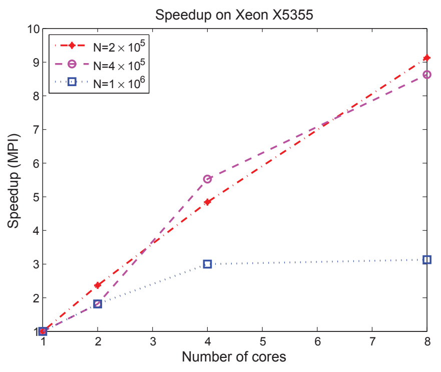
Two platforms, Intel X5355 and L5640 were used to test the effect of these optimization techniques. The block diagrams of the two platforms are shown in Figures 1.5 and 1.6. Three problem sizes were chosen: $N = 2 \times 10^5$, $4 \times 10^5$, $10^6$. On X5355, the simulation time was reduced to 1/3 of the baseline's time usage. On L5640, the simulation time was almost halved.

For OpenMP implementation, we should pay attention to the initialization of arrays on L5640 since it is a NUMA system. The MPI speedup is listed in Figure 2.1. On X5355, the speedup decreases dramatically for $N = 10^6$ since the performance is determined by the memory bandwidth.

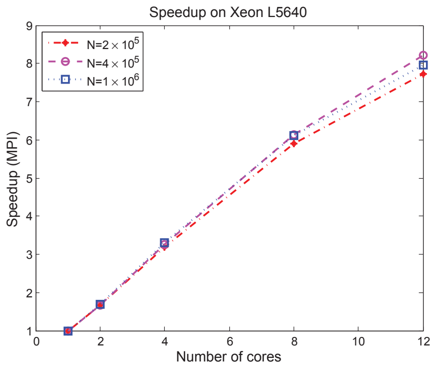
## 2.2 Paper II

Paper II is an extension of paper I. Different optimization techniques and their effect were shown in paper I. Quantitative analysis of these techniques were presented in paper II. The methodology can also be used for analyzing parallel implementations.

Different optimization techniques have the same floating point operations but different data traffic volume. For a given problem size $N$, we dissect $N$ into four stages: data in L1, L2, L3 cache, and main memory. The total simulation time is the summation of time at different



(a)



(b)

**Figure 2.1:** *Speedup on Intel Xeon X5355 and L5640.*

stages. Parallel implementation adopts the same philosophy but only has aggregated cache, memory bandwidths.

We used the fractional Adams-Bashforth-Moulton method as the test bed to check the quality of our performance models. Three platforms were used: Intel Xeon E5504, E5-2670 and AMD Opteron 6276. First, we used PAPI-v5.2.0 on AMD Opteron 6276 to verify our model. All codes were compiled using Cray C, GCC, and ICC with -O2 optimization. We compare the PAPI's events `FP_OPS`, `L1_DCA`, and `L2_DCA` with our predictions. The prediction of floating point operations and L2 data cache accesses had good agreement with PAPI results. There was nuance in the L1 accesses. We assumed that different compilers had different data reuse rate. Second, we ran the codes with $N = 2 \times 10^5$, $1 \times 10^6$, and $4 \times 10^6$. For $N = 2 \times 10^5$, all data were located in caches. Experiments showed that the models could estimate the time usage of both serial and parallel implementations.

Our model only focused on the data traffic volume. Other factors determining the code performance like the number of registers, cache misses, and communication cost were not considered. These factors affected the accuracy of the performance model. For instance, deeper loop unrolling will lead to longer code and need more registers. Therefore, there will be more L1 data cache accesses and performance degradation. The collective communication in MPI implementation is expensive when there is a number of processes. The software Scalasca showed that on Intel Xeon E5-2670, when using 16 cores, the time consumption of `MPI_Allreduce` accounted for 20.8 % of the total time usage.

## 2.3 Paper III

In paper III, time fractional heat equations is considered. We used a model problem and showed fractional heat equations could predict negative absolute temperatures.

The classical parabolic heat equation based on Fourier' law assumes infinite heat propagation speed. This assumption is inaccurate for short time, extremely low temperatures. Under these conditions, we should consider the finite heat propagation speed, which can be modelled by hyperbolic heat equations.

The fractional order equation is a generalization of the classical equation and a useful tool for describing non-Fourier heat conduction. In paper III, we considered a one-dimensional model problem where two cold waves collided in a layer. The fractional generalized Cattaneo equations, parabolic and hyperbolic heat equations were chosen to describe the problem. For the convenience of analysis, all equations were written in nondimensional forms. We solved the equations using finite difference method. The fractional order $\alpha$ was divided into two ranges: $0 < \alpha < 1$ and $1 < \alpha < 2$. For $0 < \alpha < 1$, the approximation of fractional derivative is described in Eq. (1.37). It has two values of the unknown $\mathsf{u}$. For $1 < \alpha < 2$, the approximation can be found in [30] and has three values of $\mathsf{u}$. Therefore the simulation time for $1 < \alpha < 2$ is longer than $0 < \alpha < 1$.

In the numerical experiments, we chose the nondimensional domain lengths $L^* = 1$ and 5 in order to consider two cases where the domain length was larger or smaller than phonon's mean free path. By comparing the results of different equations, we concluded that like hyperbolic heat equation, fractional heat equations may also give negative absolute temperatures. For $L^* = 1$, the generalized Cattaneo equations had the tendency of giving negative temperature. The reason is the continuum hypothesis upon which the equations depend is probably invalid. The values of fractional order also determine whether fractional heat equations could give negative absolute temperatures. However, there is no clear physical explanation of the values. We also pointed out that one solution to avoid unphysical solutions was introducing non-linear fractional equations.

## 2.4 Paper IV

In paper IV, we considered the possible biased estimation of shear modulus in media with power law characteristic.

A fractional wave equation is used to describe shear wave propagation in viscoelastic media. The equation can be derived from fractional Kelvin-Voigt model and has two frequency regimes. The shear wave is generated by a cylindrically symmetric beam. All Parameters like the fractional order, characteristic beam radius, pulse duration time, shear wave speed, viscosity, were obtained from existing literatures. The equation was solved using finite difference scheme in cylindrical coordinate. We used the fractional order $\alpha = 0.36$, 1 and 1.7 in the experiments and compared the displacement at the focal point. It was shown that the displacement at the focal point varied with different fractional order. Specifically, for $0 < \alpha < 1$, the equation predicted higher displacement and smaller shear modulus than the classical model. The opposite situation applied for $1 < \alpha < 2$. The relation between focal displacement and fractional order between 0 and 2 is shown in Figure 2.2.

We also used two case to show that different combinations of fractional order and shear wave propagation speed could give the same displacement even if the actual shear moduli were different. We concluded that estimating shear modulus by the amplitude at the focal point might not be reliable if different equations were used. The time-to-peak method was considered for the fractional order equation. Experiments showed this method was valid only for small fractional order.

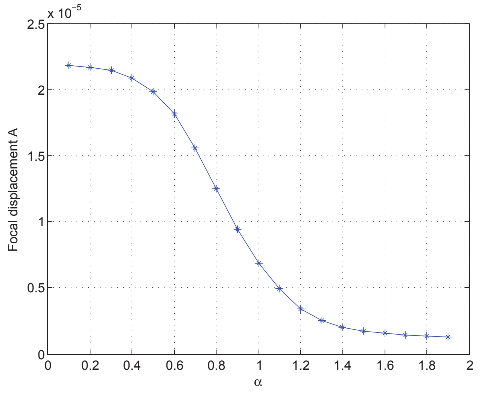


**Figure 2.2:** *The displacement at the focal point changes with the fractional order $\alpha$.*

## 2.5 Paper V

In paper V, we considered the optimization and parallelization of solving time-fractional diffusion equations using a finite difference method. The performance model and scalability were also discussed.

At present, the finite difference method is frequently used for solving factional differential equations numerically. Code optimization, parallelization, performance analysis and scalability of finite difference schemes are important topics worthy investigation. The serial code optimization was achieved using loop unrolling in time with depth 2 and 4. Further performance enhancement was obtained using Advanced Vector Extensions (AVX) instructions. The parallelization within a computing node were enabled via MPI or OpenMP. For 3D problems with a large size, more computing nodes were needed for storing the large numbers of data. Hybrid programming with OpenMP and MPI was used for parallelization. More specifically, MPI was used for communication among the computing nodes while OpenMP managed the workload within each node.

Scalability was an important topic in this paper. It reveals how the simulation time varies as both the problem size and number of processors increase proportionally. For the 3D problem, we increased both the problem size and the number of computing nodes simultaneously, the simulation time almost remained the same. This demonstrated that when solving 3D fractional differential equations, the finite difference method had a good property of scaling. Combing the performance model and scaling property, we can estimate the simulation time of solving 3D problems with a much larger size.

# Chapter 3

# Future work

This chapter summarizes the main contributions of the thesis and shows possible future research directions.

The main contributions of the thesis are

- Show that the simulation time for solving fractional differential equations can be reduced by using optimization techniques like *loop fusion*, *loop unrolling* and *alternating direction of loop traversal*.

- Introduce serial and parallel performance models. We give quantitative analysis of aforementioned techniques and put up models which give deep understanding about the achievable performance. The models can predict simulation time, evaluate code implementations, and pinpoint the bottleneck of computer systems. They can be extended to similar numerical method for approximation of convolution.

- Show that fractional heat equations can give negative absolute temperatures.

- Simulate shear wave propagation in viscoelastic media and demonstrate that different wave equations may lead to biased estimation of shear modulus.

- Develop parallel simulator for solving 3D fractional diffusion equation and show that the finite difference method has good property of weak scaling when solving 3D fractional differential equations.

There are still many interesting questions worthy investigating in the future.

- **Developing nonlinear fractional heat equation**
  Paper III shows that fractional heat equations may give temperatures below absolute zero for small domain length. An natural extension of our study is to modify fractional heat equations to remedy this problem. In paper III, the thermal conductivity is a constant. From physical point of view, when temperature approaches to zero, the heat capacity is strongly temperature-dependent. Consequently, the heat equation will be nonlinear. The future work should be deriving a nonlinear equation which has temperature-dependent thermal conductivity and satisfies the second law of thermodynamics.

- **Using finite element method to solve fractional differential equation in irregular domains**
  In this thesis, our application examples are only confined within spatial domain of a regular shape. However, real-world problems usually have complicated geometries or boundaries. For instance, it is difficult to model shear wave propagation in arterial using finite difference method. Finite element method (FEM) is a powerful tool to deal with this difficulty. A possible work is to use fractional wave equation to model shear wave propagation in artery and use FEM to get numerical results.

- **Parallelization of spatial fractional derivative solvers**
  All equations in this thesis only include time fractional derivative operator(s). Fractional models in space or time-space are also widely used in diffusion theory and elastography [25]. One particular topic is developing parallel solvers which can solve fractional differential equations with spatial fractional derivatives.

- **Developing a numerical software package**
  Developing numerical simulator for fractional differential equation is a complex and time consuming task. An interesting work in the future is developing an easy-to-use and high quality software. It contains a common set of generic numerical routines for solving linear and nonlinear fractional differential equations. It also has parallel toolbox and visualization support. Therefore, researchers can save their time and refine their results.

# Bibliography

[1] L. Adhianto and B. Chapman. Performance modeling of communication and computation in hybrid MPI and OpenMP applications. *Simul. Model Pract. Th.*, 15(4):481–491, 2007.

[2] K. Adolfsson, M. Enelund, and S. Larsson. Adaptive discretization of fractional order viscoelasticity using sparse time history. *Comput. Methods in Appl. Mech. Eng.*, 193(42):4567–4590, 2004.

[3] R. Aversa, B. Di Martino, M. Rak, S. Venticinque, and U. Villano. Performance prediction through simulation of a hybrid MPI/OpenMP application. *Parallel Comput.*, 31(10):1013–1033, 2005.

[4] C. Bai and A. S. Lavine. On hyperbolic heat conduction and the second law of thermodynamics. *J. Heat Transfer*, 117(2):256–263, 1995.

[5] D. Baleanu, K. Diethelm, E. Scalas, and J. J. Trujillo. *Fractional Calculus: Models and Numerical Methods*. World Scientific, Boston, 2012.

[6] S. Bentil and R. Dupaix. Exploring the mechanical behavior of degrading swine neural tissue at low strain rates via the fractional Zener constitutive model. *J. Mech. Behav. Biomed. Mater.*, 30:83–90, 2014.

[7] M. Caputo, J. Carcione, and F Cavallini. Wave simulation in biologic media based on the Kelvin-Voigt fractional-derivative stress-strain relation. *Ultrasound Med. Biol.*, 37(6):996–1004, 2011.

[8] A. Carella. *Spectral Finite Element Methods for solving Fractional Differential Equations with applications in Anomalous Transport*. PhD thesis, Norwegian University of Science and Technology, 2012.

[9] C. Cattaneo. Sulla conduzione de calore (On the conduction of heat). *Atti del Semin. Mat. e Fis. Univ. Modena*, (3):83–101, 1948.

[10] W. Chen and S. Holm. Fractional Laplacian time-space models for linear and nonlinear lossy media exhibiting arbitrary frequency power-law dependency. *J. Acoust. Soc. Am.*, 115:1424–1430, 2004.

[11] A. Compte and R. Metzler. The generalized Cattaneo equation for the description of anomalous transport processes. *J. Phys. A: Math. Gen.*, 30(21):7277–7289, 1997.

[12] W. Deng. Short memory principle and a predictor–corrector approach for fractional differential equations. *J. Comput. Appl. Math.*, 206(1):174–188, 2007.

[13] W. Deng. Finite element method for the space and time fractional Fokker-Planck equation. *SIAM J. Numer. Anal.*, 47(1):204–226, 2008.

[14] K. Diethelm. *The Analysis of Fractional Differential Equations*. Springer, 2010.

[15] K. Diethelm. An efficient parallel algorithm for the numerical solution of fractional differential equations. *Fract. Calc. Appl. Anal.*, 14:475–490, 2011.

[16] K. Diethelm, N. J. Ford, and A. D. Freed. A predictor-corrector approach for the numerical solution of fractional differential equations. *Nonlinear Dynam.*, 29:3–22, 2002.

[17] N. J. Ford and A. C. Simpson. The numerical solution of fractional differential equations: speed versus accuracy. *Numer. Algorithms*, 26(4):333–346, 2001.

[18] N. J. Ford, J. Xiao, and Y. Yan. A finite element method for time fractional partial differential equations. *Fract. Calc. Appl. Anal.*, 14(3):454–474, 2011.

[19] H. Ghazizadeh, A. Azimi, and M. Maerefat. An inverse problem to estimate relaxation parameter and order of fractionality in fractional single-phase-lag heat equation. *Int. J. Heat Mass Transfer*, 55(7):2095–2101, 2012.

[20] R. Gorenflo, Y. Luchko, and F. Mainardi. Wright functions as scale-invariant solutions of the diffusion-wave equation. *J. Comput. Appl. Math.*, 118(1):175–191, 2000.

[21] P. Gupta and M. Singh. Homotopy perturbation method for fractional Fornberg–Whitham equation. *Comput. Math. Appl.*, 61(2):250–254, 2011.

[22] M. E. Gurtin and A. C. Pipkin. A general theory of heat conduction with finite wave speeds. *Arch. Rational Mech. Anal.*, 31(2):113–126, 1968.

[23] G. Hager and G. Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, 2010.

[24] R. Hilfer. *Applications of Fractional Calculus in Physics*. World Scientific, Singapore, 2000.

[25] S. Holm and S. P. Näsholm. Comparison of fractional wave equations for power law attenuation in ultrasound and elastography. *Ultrasound Med. Biol.*, 40(4):695–703, 2014.

[26] S. Holm, S. P. Näsholm, F. Prieur, and R. Sinkus. Deriving fractional acoustic wave equations from mechanical and thermal constitutive equations. *Comput. Math. Appl.*, 66(5):621–629, 2013.

[27] S. Holm and R. Sinkus. A unifying fractional wave equation for compressional and shear waves. *J. Acoust. Soc. Am.*, 127:542–548, 2010.

[28] A. A. Kilbas, H. M. Srivastava, and J. J. Trujillo. *Theory and Applications of Fractional Differential Equations*, volume 204. Elsevier, Amsterdam, 2006.

[29] C. Li and W. Deng. Remarks on fractional derivatives. *Appl. Math. Comput.*, 187(2):777–784, 2007.

[30] C. Li, Z. Zhao, and Y. Chen. Numerical approximation of nonlinear fractional differential equations with subdiffusion and superdiffusion. *Comput. Math. Appl.*, 62:855–875, 2011.

[31] R. Lin and F. Liu. Fractional high order methods for the nonlinear fractional ordinary differential equation. *Nonlinear Anal. Theor.*, 66(4):856–869, 2007.

[32] F. Liu, V.V. Anh, I. Turner, and P. Zhuang. Time fractional advection-dispersion equation. *J. Comput. Appl. Math.*, 13(1):233–245, 2003.

[33] Y. Luchko and A. Punzi. Modeling anomalous heat transport in geothermal reservoirs via fractional diffusion equations. *Int. J. Geomath.*, 1(2):257–276, 2011.

[34] T. Machado, V. Kiryakova, and F. Mainardi. Recent history of fractional calculus. *Commun. Nonlinear Sci. Numer. Simul.*, 16(3):1140–1153, 2011.

[35] F. Mainardi. *Fractals and Fractional Calculus in Continuum Mechanics*. Springer Verlag, 1997.

[36] M. M. Meerschaert, H. Scheffler, and C. Tadjeran. Finite difference methods for two-dimensional fractional dispersion equation. *J. Comput. Phys.*, 211(1):249–261, 2006.

[37] M. M. Meerschaert and C. Tadjeran. Finite difference approximations for fractional advection–dispersion flow equations. *J. Comput. Appl. Math.*, 172(1):65–77, 2004.

[38] R. Metzler and J. Klafter. The random walk's guide to anomalous diffusion: a fractional dynamics approach. *Phys. Rep.*, 339(1):1–77, 2000.

[39] K. S. Miller and B. Bertram. *An introduction to the fractional calculus and fractional differential equations*. John Wiley & Sons Inc., 1993.

[40] S. Momani and Z. Odibat. Analytical solution of a time-fractional Navier–Stokes equation by Adomian decomposition method. *Appl. Math. Comput.*, 177(2):488–494, 2006.

[41] A. I. Nachman, J. F. Smith, and R. C. Waag. An equation for acoustic propagation in inhomogeneous media with relaxation losses. *J. Acoust. Soc. Am.*, 88:1584, 1990.

[42] K. B. Oldham and J. Spanier. *The Fractional Calculus*. Academic press, New York, 1974.

[43] I. Podlubny. *Fractional Differential Equations*. Academic Press, San Diego, CA, 1999.

[44] I. Podlubny, A. Chechkin, T. Skovranek, Y. Chen, and B. M. Vinagre Jara. Matrix approach to discrete fractional calculus II: Partial fractional differential equations. *J. Comput. Phys.*, 228(8):3137–3153, 2009.

[45] Y. Z. Povstenko. Fundamental solutions to three-dimensional diffusion-wave equation and associated diffusive stresses. *Chaos Solitons Fractals*, 36(4):961–972, 2008.

[46] T. Rauber and G. Rünger. *Parallel Programming: For Multicore and Cluster Systems*. Springer, 2010.

[47] M. B. Rubin. Hyperbolic heat conduction and the second law. *Int. J. Eng. Sci.*, 30(11):1665–1676, 1992.

[48] S. G. Samko, A. A. Kilbas, and O. I. Marichev. *Fractional Integrals and Derivatives: Theory and Applications*. Gordon and Breach, New York, 1993.

[49] R. Sinkus, K. Siegmann, T. Xydeas, M. Tanter, C. Claussen, and M. Fink. MR elastography of breast lesions: Understanding the solid/liquid duality can improve the specificity of contrast-enhanced MR mammography. *Magn. Res. in Med.*, 58(6):1135–1144, 2007.

[50] E. Sousa. Finite difference approximations for a fractional advection diffusion problem. *J. Comput. Phys.*, 228(11):4038–4054, 2009.

[51] The STREAM2 Home Page. http://www.cs.virginia.edu/stream/stream2/.

[52] H. Sun, Y. Zhang, W. Chen, and D. M. Reeves. Use of a variable-index fractional-derivative model to capture transient dispersion in heterogeneous media. *J. Contam. Hydrol.*, 157:47–58, 2014.

[53] T. L. Szabo. Time domain wave equations for lossy media obeying a frequency power law. *J. Acoust. Soc. Am.*, 96:491, 1994.

[54] P. J. Torvik and R. L. Bagley. On the appearance of the fractional derivative in the behavior of real materials. *J. Appl. Mech.*, 51(2):294–298, 1984.

[55] Q. Wang. Homotopy perturbation method for fractional KdV equation. *Appl. Math. Comput.*, 190(2):1795–1802, 2007.

[56] W. Wei. *Effective Use of Multicore-based Parallel Computers for Scientific Computing*. PhD thesis, University of Oslo, 2012.

[57] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[58] M. G. Wismer and R. Ludwig. An explicit numerical time domain formulation to simulate pulsed pressure waves in viscous fluids exhibiting arbitrary frequency power law attenuation. *IEEE Trans. Ultrason. Ferroelectr., Freq. Control*, 42(6):1040–1049, 1995.

[59] X. Wu and V. Taylor. Performance modeling of hybrid MPI/OpenMP scientific applications on large-scale multicore supercomputers. *J. Comput. Syst. Sci.*, 79(8):1256–1268, 2013.

[60] M Zhang, P. Nigwekar, B. Castaneda, K. Hoyt, J. V. Joseph, A. di Sant'Agnese, E. M Messing, J. G. Strang, D. J. Rubens, and K. J. Parker. Quantitative characterization of viscoelastic properties of human prostate correlated with histology. *Ultrasound Med. Biol.*, 34(7):1033–1042, 2008.

# Paper I:

# Efficient Implementations of the Adams-Bashforth-Moulton Method for Solving Fractional Differential Equations

# Efficient Implementations of the Adams-Bashforth-Moulton Method for Solving Fractional Differential Equations

**Wei Zhang** [*] **Xing Cai** [**,*]

[*] *Department of Informatics, University of Oslo, Norway (e-mail: weizhang@student.matnat.uio.no)*
[**] *Simula Research Laboratory, Norway (e-mail: xingca@simula.no)*

**Abstract:** The second-order Adams-Bashforth-Moulton scheme is a predictor-corrector approach to solving initial-value problems that involve fractional derivatives. The number of required floating-point operations grows as $\mathcal{O}(N^2)$, where $N$ is the number of time steps. This potentially huge amount of computation, together with the large memory requirement of order $\mathcal{O}(N)$, calls for efficient software implementations. In this context, we will demonstrate the importance of code optimization techniques like loop fusion, loop unrolling and alternating directions for loop traversal. On the basis of efficient serial implementations, high-performance parallel implementations, via MPI and OpenMP programming, can be easily developed for modern multi-core processors. The particular Bagley-Torvik equation, which can be formulated as a system of four coupled fractional differential equations, is used as a test bed for studying the obtainable computing speeds.

*Keywords:* System of fractional differential equations, Adams-Bashforth-Moulton method, Bagley-Torvik equation, loop optimization, parallel computing, multi-core processor.

## 1. INTRODUCTION

Numerical methods for solving fractional differential equations often involve convolutions over the entire temporal and/or spatial domains. Special care is needed when implementing computer codes for such algorithms, because good data reuse will decrease the volume of data traffic from the main memory and increase data-hit ratios in the caches, thus providing sustainable performance not far away from a computer's theoretical peak. In this paper, we choose the Adams-Bashforth-Moulton method, see e.g. Diethelm et al. (2002), as a representative numerical scheme. It will be shown that techniques of loop optimization can greatly improve the performance of a naively implemented serial code, while also laying the foundation for high-performance parallel implementations. Numerical experiments in connection with the Bagley-Torvik equation, when formulated as a system of four coupled fractional differential equations, will demonstrate the computing speed gains.

## 2. MATHEMATICAL MODEL

We consider in this paper the following system of $m$ coupled scalar fractional differential equations:

$$\begin{cases} D_*^\alpha y_1(t) = & g_1(y_1(t), y_2(t), \ldots, y_m(t), t), \\ D_*^\alpha y_2(t) = & g_2(y_1(t), y_2(t), \ldots, y_m(t), t), \\ \quad\vdots & \quad\quad\vdots \\ D_*^\alpha y_m(t) = & g_m(y_1(t), y_2(t), \ldots, y_m(t), t), \end{cases} \tag{1}$$

where $D_*^\alpha$ denotes a fractional differential operator that is defined in the sense of Caputo:

$$D_*^\alpha y := J^{\lceil\alpha\rceil-\alpha} D^{\lceil\alpha\rceil} y, \tag{2}$$

where $D^{\lceil\alpha\rceil}$ is the classical differential operator of integer order $\lceil\alpha\rceil$, and $J^\mu$ denotes the Riemann-Liouville integral operator of order $\mu > 0$, defined by

$$J^\mu y(x) = \frac{1}{\Gamma(\mu)} \int_0^x \frac{y(t)}{(x-t)^{1-\mu}} \, \mathrm{d}t, \tag{3}$$

where $\Gamma$ denotes Euler's Gamma function. For simplicity, we assume $0 < \alpha < 1$, so each scalar fractional differential equation in (1) is equipped with one initial condition.

For the convenience of presenting the numerical algorithm in the next section, we will also adopt a compact representation of system (1) as follows:

$$D_*^\alpha \boldsymbol{Y}(t) = \boldsymbol{g}(\boldsymbol{Y}(t), t), \tag{4}$$

where $\boldsymbol{Y}(t) = (y_1(t), y_2(t), \ldots, y_m(t))^T$ is the solution vector that we seek and $\boldsymbol{g} = (g_1, g_2, \ldots, g_m)^T$.

## 3. ADAMS-BASHFORTH-MOULTON SCHEME

By using a fixed time stepping size $h$, we can denote by

$$\boldsymbol{Y}_i = (y_1^i, y_2^i, \ldots, y_m^i)^T$$

the numerical approximation of $\boldsymbol{Y}$ at a discrete time level $t = ih$, i.e.,

$$(y_1(ih), y_2(ih), \ldots, y_m(ih))^T.$$

The second-order Adams-Bashforth-Moulton scheme is an iterative procedure, meaning that when $\boldsymbol{Y}_0, \boldsymbol{Y}_1, \ldots, \boldsymbol{Y}_j$ are known, the numerical solution for the next time level, $\boldsymbol{Y}_{j+1}$, arises from a predictor-corrector step. More specifically, the so-called predictor $\boldsymbol{Y}^P_{j+1}$ is first computed as

$$\boldsymbol{Y}^P_{j+1} = \boldsymbol{Y}_0 + h^\alpha \sum_{k=0}^{j} b_{j-k}\boldsymbol{g}(\boldsymbol{Y}_k, kh), \tag{5}$$

where the weights $b_\mu$ in (5) are given by

$$b_\mu = \frac{(\mu+1)^\alpha - \mu^\alpha}{\Gamma(\alpha+1)}. \tag{6}$$

Then, $\boldsymbol{Y}_{j+1}$ is computed by a corrector step, namely,

$$\boldsymbol{Y}_{j+1} = \boldsymbol{Y}_0 + h^\alpha\, c_j \boldsymbol{g}(\boldsymbol{Y}_0, 0) + h^\alpha \left( \sum_{k=1}^{j} a_{j-k}\boldsymbol{g}(\boldsymbol{Y}_k, kh) + \frac{\boldsymbol{g}(\boldsymbol{Y}^P_{j+1}, (j+1)h)}{\Gamma(\alpha+2)} \right), \tag{7}$$

where the weights $a_\mu$ and $c_j$ in (7) are given by

$$a_\mu = \frac{(\mu+2)^{\alpha+1} - 2(\mu+1)^{\alpha+1} + \mu^{\alpha+1}}{\Gamma(\alpha+2)}, \tag{8}$$

$$c_j = \frac{j^{\alpha+1} - (j-\alpha)(j+1)^\alpha}{\Gamma(\alpha+2)}. \tag{9}$$

*Observations.* If $N$ denotes the total number of time steps used, the total number of floating-point operations needed by the Adams-Bashforth-Moulton scheme grows as $\mathcal{O}(N^2)$. The majority of the floating-point operations are multiplications and additions involved in the two summations in (5) and (7), one related to $b_\mu$ and the other related to $a_\mu$. In addition, the evaluations of $\boldsymbol{g}(\boldsymbol{Y}_k, kh)$ may also involve a substantial number of floating-point operations. Moreover, all the previously computed numerical solutions $\boldsymbol{Y}_1, \boldsymbol{Y}_2, \ldots, \boldsymbol{Y}_{N-1}$ have to be stored for computing $\boldsymbol{Y}_N$.

## 4. THE BAGLEY-TORVIK EQUATION

As a concrete example of system (1), we will use the Bagley-Torvik equation, see Torvik and Bagley (1984):

$$Ay''(t) + BD_*^{3/2}y(t) + Cy(t) = f(t), \tag{10}$$

where $A$, $B$, and $C$ are scalar constants. Equation (10) can be rewritten as a system of four coupled scalar fractional differential equations with $\alpha = 1/2$, see Diethelm and Ford (2002). More specifically, we have

$$\begin{cases} D_*^{1/2}y_1(t) = y_2(t), \\ D_*^{1/2}y_2(t) = y_3(t), \\ D_*^{1/2}y_3(t) = y_4(t), \\ D_*^{1/2}y_4(t) = A^{-1}(-Cy_1(t) - By_4(t) + f(t)), \end{cases} \tag{11}$$

with initial conditions

$$y_1(0) = y_0, \quad y_2(0) = 0, \quad y_3(0) = y_0', \quad y_4(0) = 0.$$

We also note that evaluating $g_i(y_1(t), y_2(t), y_3(t), y_4(t), t)$ is straightforward (no floating-point operations needed) for $i = 1, 2, 3$, whereas evaluating $g_4(y_1(t), y_2(t), y_3(t), y_4(t), t)$ may require quite a few floating-point operations, depending on the particular form of $f(t)$.

## 5. IMPLEMENTATIONS

We use the C programming language, see Kernighan and Ritchie (1988), to implement the Adams-Bashforth-Moulton scheme for solving (11). As the basic data structure, four arrays y1, y2, y3, and y4 are used to store the numerical solutions of $y_1(t)$, $y_2(t)$, $y_3(t)$, and $y_4(t)$ at $N$ discrete time levels. In addition, one array with name a is used to store the pre-calculated weights $a_\mu$ needed in (7), and another array with name b is for the pre-calculated weights $b_\mu$ needed in (5). Moreover, we have used an additional array, named g4yt, to store the evaluated values of

$$g_4(\texttt{y1[i],y2[i],y3[i],y4[i],i*h}) \quad 0 \le i < N.$$

The reason for adopting the g4yt array is to avoid repeated evaluations of $g_4(y_1^i, y_2^i, y_3^i, y_4^i, ih)$, which may be costly.

### *5.1 Baseline: a Naive Implementation*

The following implementation naively follows the mathematical definition of the Adams-Bashforth-Moulton scheme (5)-(9). An outer for-loop with index j is for the time stepping. In each j iteration, one for-loop is used to compute $\boldsymbol{Y}^P_{j+1}$ according to (5), and another for-loop is used to compute $\boldsymbol{Y}_{j+1}$ following (7).

```
for (j=0; j<N; j++) {
  sum_b1 = sum_b2 = sum_b3 = sum_b4 = 0.;
  for (k=0; k<=j; k++) {
    sum_b1 += b[j-k]*y2[k];
    sum_b2 += b[j-k]*y3[k];
    sum_b3 += b[j-k]*y4[k];
    sum_b4 += b[j-k]*g4yt[k];
  }

  /* predictor values */
  y1pred = y1[0]+h_alpha*sum_b1;
  y2pred = y2[0]+h_alpha*sum_b2;
  y3pred = y3[0]+h_alpha*sum_b3;
  y4pred = y4[0]+h_alpha*sum_b4;

  c_j = (pow(j,alpha+1)-(j-alpha)*pow(j+1,alpha))/gamma2;
  sum_a1 = c_j*y2[0];
  sum_a2 = c_j*y3[0];
  sum_a3 = c_j*y4[0];
  sum_a4 = c_j*g4yt[0];
  for (k=1; k<=j; k++) {
    sum_a1 += a[j-k]*y2[k];
    sum_a2 += a[j-k]*y3[k];
    sum_a3 += a[j-k]*y4[k];
    sum_a4 += a[j-k]*g4yt[k];
  }

  y1[j+1] = y1[0]+h_alpha*(sum_a1+y2pred/gamma2);
  y2[j+1] = y2[0]+h_alpha*(sum_a2+y3pred/gamma2);
  y3[j+1] = y3[0]+h_alpha*(sum_a3+y4pred/gamma2);
  t = (j+1)*h;
  y4[j+1] = y4[0]+h_alpha*(sum_a4
            +g4(y1pred,y2pred,y3pred,y4pred,t)/gamma2);
  g4yt[j+1] = g4(y1[j+1],y2[j+1],y3[j+1],y4[j+1],t);
}
```

It should be noted that values of arrays a and b are computed beforehand once and for all. The variable h_alpha contains the constant value $h^\alpha$.

The number of floating-point operations that can be carried out per memory read (or write) is an important metric for the efficiency of a computer code. For the above baseline version of implementation, within the two for-loops that both use index k, we can see that each k iteration requires five memory reads:

(1) b[j-k] or a[j-k]

(2) `y2[k]`
(3) `y3[k]`
(4) `y4[k]`
(5) `g4yt[k]`

Correspondingly, the number of floating-point operations executed per `k` iteration is eight (four multiplications and four additions). Therefore, this results in an average of 8/5 floating-point operations per memory read.

### *5.2 Improvement 1: Loop Fusion*

In the baseline version of the implementation, the values of `y2[k]`, `y3[k]`, `y4[k]`, and `g4yt[k]` ($0 \le k \le j$) are loaded twice per `j` iteration. This is because of the two separate `k`-indexed `for`-loops. To improve the ratio of floating-point operations versus memory reads, we can combine the two loops into one. This is a standard optimization technique called *loop fusion*, see e.g. Goedecker and Hoisie (2001), which results in the following implementation:

```
for (j=0; j<N; j++) {
  sum_b1 = b[j]*y2[0];
  sum_b2 = b[j]*y3[0];
  sum_b3 = b[j]*y4[0];
  sum_b4 = b[j]*g4yt[0];

  c_j = (pow(j,alpha+1)-(j-alpha)*pow(j+1,alpha))/gamma2;
  sum_a1 = c_j*y2[0];
  sum_a2 = c_j*y3[0];
  sum_a3 = c_j*y4[0];
  sum_a4 = c_j*g4yt[0];

  for (k=1; k<=j; k++) {
    sum_b1 += b[j-k]*y2[k];
    sum_b2 += b[j-k]*y3[k];
    sum_b3 += b[j-k]*y4[k];
    sum_b4 += b[j-k]*g4yt[k];
    sum_a1 += a[j-k]*y2[k];
    sum_a2 += a[j-k]*y3[k];
    sum_a3 += a[j-k]*y4[k];
    sum_a4 += a[j-k]*g4yt[k];
  }

  /* remaining part same as the baseline implementation */
}
```

A new count shows that, for the above implementation that is based on loop fusion, the ratio of floating-point operations versus memory reads is now 16/6.

### *5.3 Improvement 2: Loop Unrolling*

Each `j` iteration, in both the versions of baseline and loopfusion, computes $\boldsymbol{Y}_{j+1}$ in form of `y1[j+1]`, `y2[j+1]`, `y3[j+1]`, and `y4[j+1]`. As another way of improving the performance, we can adopt the technique of *loop unrolling*, see e.g. Goedecker and Hoisie (2001). The result is that the number of `j` iterations is halved, but each `j` iteration instead computes both $\boldsymbol{Y}_{j+1}$ and $\boldsymbol{Y}_{j+2}$. Here, we skip the lengthy code listing of this version and only mention that the ratio of floating-point operations versus memory reads is increased to 32/8.

### *5.4 Improvement 3: Alternating Loop Traversal*

There is a third possibility for performance improvement that can be applied to both the versions of loopfusion and loopunrolling. More specifically, instead of running one `k`-indexed `for`-loop in each `j` iteration, two `k`-indexed `for`-loops can be executed one after another. (Of course, the number of `j` iterations has to be halved accordingly.) The first `for`-loop goes in the increasing order of the `k` index, and the second `for`-loop goes in the opposite direction with respect to `k`. Although the ratio of floating-point operations versus memory reads is not affected, such a trick is good with respect to the temporal data locality in the caches, as we alternate between "forward" and "backward" `for`-loops. This technique (let us call it *forward-backward*) can have a positive performance impact when the footprint of the data structure exceeds the caches' capacity.

### *5.5 Performance Comparison of Serial Implementations*

We have tested four serial implementations for three problem sizes, $N = 2 \times 10^5, 4 \times 10^5, 10^6$. Two hardware platforms were used, one being Intel's Clovertown 2.66 GHz X5355 quad-core CPU, the other Intel's Westmere-EP 2.26 GHz L5640 hexa-core CPU. On the X5355 CPU, a pair of cores share 4MB of L2 cache, whereas four cores on the L5640 CPU share 12MB of L3 cache. The serial C compiler used on the Clovertown system is `icc` of version 11.1 with optimization flag `-O2`. On the Westmere-EP system, the C compiler used is `gcc` of version 4.1 also with optimization flag `-O2`. All the computations used for this paper adopted double precision, i.e., each floating-point value is of type `double`.

Table 1. Serial time usages (in seconds) measured on multi-core CPUs

| Clovertown 2.66GHz X5355 CPU (using 1 core) | | | |
|---|---|---|---|
| $N$ | $2 \times 10^5$ | $4 \times 10^5$ | $1 \times 10^6$ |
| Baseline | 260.10 | 1545.08 | 10760.8 |
| Loopfusion | 167.75 | 911.34 | 6653.4 |
| Loopunrolling | 99.14 | 484.96 | 3445.7 |
| Unroll+FwBw | 86.02 | 423.12 | 3189.2 |
| **Westmere-EP 2.26GHz L5640 CPU (using 1 core)** | | | |
| $N$ | $2 \times 10^5$ | $4 \times 10^5$ | $1 \times 10^6$ |
| Baseline | 136.6 | 584.6 | 4253.6 |
| Loopfusion | 91.1 | 388.6 | 2639.4 |
| Loopunrolling | 84.9 | 345.3 | 2202.2 |
| Unroll+FwBw | 93.4 | 375.8 | 2392.2 |

From Table 1, we can see that all the three performance improvement strategies have positive effects, except for the forward-backward technique on the Westmere-EP system.

The minimum total number of floating-point operations needed is $8N^2$, i.e., when we only count those involved in the `k`-indexed `for`-loops. The smaller the problem size, the better the data-hit ratio in the caches. Therefore, the best performance (in terms of floating-point operations per second) was associated with $N = 2 \times 10^5$. On one core of the Clovertown X5355 CPU, the highest sustained floating-point rate was $3.72 \times 10^9$ FLOP/s, calculated from 86.02 seconds being the total computing time. Similarly, for the Westmere-EP L5640 CPU, the highest single-core sustained floating-point rate was $3.77 \times 10^9$ FLOP/s.

### 5.6 Parallelization

The topic of parallelizing the Adams-Bashforth-Moulton scheme has been discussed in Diethelm (2011), where the author looked at the possibility of letting multiple CPU cores/threads concurrently compute multiple discrete time levels. We adopt, however, a more straightforward approach to parallelization in this paper. More specifically, we use multiple CPU cores/threads to divide the work involved in every `k`-indexed `for`-loop.

In our MPI-based parallelization, we compute for each MPI process

```
my_k_start = (my_rank*j)/num_procs+1;
my_k_stop = ((my_rank+1)*j)/num_procs;
```

so that the MPI process only executes the `k` iterations between `my_k_start` and `my_k_stop`. Then, the `MPI_Allreduce` function, see e.g. Gropp et al. (1994), is invoked to sum up the contributions from all the MPI processes.

In our OpenMP-enabled parallelization, we simply wrap the outer `j`-indexed `for`-loop as an OpenMP parallel region. Then, each `k`-indexed `for`-loop is preceded by, e.g., the following code annotation:

```
#pragma omp for reduction(+:sum_a1,sum_a2,sum_a3,sum_a4)
for (k=1; k<=j; k++) {
   /* same code as before */
}
```

We refer the readers to e.g. Chapman et al. (2007) for the details of OpenMP programming.

Table 2. Parallel time usages (in seconds) measured on dual-socket multi-core systems

| Dual-socket Clovertown 2.66GHz X5355 quad-core CPUs | | | |
|---|---|---|---|
| $N$ | $2 \times 10^5$ | $4 \times 10^5$ | $1 \times 10^6$ |
| 2 MPI procs | 36.67 | 233.69 | 1766.5 |
| 4 MPI procs | 17.83 | 76.68 | 1070.3 |
| 8 MPI procs | 9.43 | 49.11 | 1026.6 |
| 2 OMP threads | 37.42 | 242.46 | 2068.3 |
| 4 OMP threads | 18.15 | 81.67 | 1424.4 |
| 8 OMP threads | 9.43 | 57.32 | 1423.7 |
| **Dual-socket Westmere-EP 2.26GHz L5640 hexa-core CPUs** | | | |
| $N$ | $2 \times 10^5$ | $4 \times 10^5$ | $1 \times 10^6$ |
| 2 MPI procs | 56.3 | 224.4 | 1415.1 |
| 4 MPI procs | 29.3 | 115.4 | 726.1 |
| 8 MPI procs | 15.8 | 61.3 | 391.5 |
| 12 MPI procs | 12.1 | 45.8 | 300.2 |

In Table 2, the best MPI and OpenMP computing times are reported for the Clovertown system, which has two quad-core X5355 CPUs. Good parallel performance was obtained on this system for $N = 2 \times 10^5$ and $4 \times 10^5$. The parallel performance associated with $N = 10^6$ was less satisfactory, because for this large problem size the aggregate main memory bandwidth became the dominating performance bottleneck. For the Westmere-EP system, which has two hexa-core L5640 CPUs, we have only listed the best MPI computing times. Our OpenMP implementation actually failed to achieve any decent speedups. We suspect this to be due to the non-uniform memory access (NUMA) architecture of the Westmere-EP system, but will need thorough experiments and analyses in future to confirm our suspicion. In comparison, the Clovertown system has a UMA architecture.

## 6. CONCLUDING REMARKS

Detailed time measurements have shown that the computing speed of different implementations of the Adams-Bashforth-Moulton method is determined by the degree of data reuse. Loop optimization techniques like loop fusion, loop unrolling, and alternating direction of loop traversal can dramatically improve a naively programmed baseline serial implementation. The benefits of these performance improving techniques can, in many cases, also be transferred to MPI and OpenMP parallel implementations. A quantitative performance model needs to be developed in future, so that we can be certain whether there is space for further performance improvements.

## ACKNOWLEDGEMENTS

The first author is funded by a grant from the China Scholarship Council. A part of the computer time used for the paper has been provided by a grant from the NOTUR project of Norway.

## REFERENCES

Chapman, B., Jost, G., and van der Pas, R. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming.* MIT Press.

Diethelm, K. (2011). An efficient parallel algorithm for the numerical solution of fractional differential equations. *Fractional Calculus and Applied Analysis*, 14, 475–490.

Diethelm, K. and Ford, N.J. (2002). Numerical solution of the Bagley-Torvik equation. *BIT Numerical Mathematics*, 42, 490–507.

Diethelm, K., Ford, N.J., and Freed, A.D. (2002). A predictor-corrector approach for the numerical solution of fractional differential equations. *Nonlinear Dynamics*, 29, 3–22.

Goedecker, S. and Hoisie, A. (2001). *Performance Optimization of Numerically Intensive Codes.* SIAM.

Gropp, W., Lusk, E., and Skjellum, A. (1994). *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press.

Kernighan, B. and Ritchie, D. (1988). *The C Programming Language.* Prentice-Hall.

Torvik, P.J. and Bagley, R.L. (1984). On the appearance of the fractional derivative in the behavior of real materials. *Journal of Applied Mechanics*, 294–298.

# Paper II:

# Performance Modeling of Serial and Parallel Implementations of the Fractional Adams-Bashforth-Moulton Method

# PERFORMANCE MODELING OF SERIAL AND PARALLEL IMPLEMENTATIONS OF THE FRACTIONAL ADAMS-BASHFORTH-MOULTON METHOD

**Wei Zhang**[1], **Wenjie Wei**[2], **Xing Cai**[1,2]

## Abstract

Numerical schemes for solving fractional differential equations are computationally heavy, due to the floating-point operations needed and, more importantly, the data flow within the entire memory system of a computer. We choose the fractional Adams-Bashforth-Moulton method as a representative numerical scheme, and review various code optimizations that can be applied to its serial and parallel implementations. As the most important contribution of this paper, we propose a simple methodology to analyze the achievable serial and parallel performance, based on quantifying the amount of data flow through various stages of the entire memory system, together with a small set of easily obtainable hardware parameters. This quantitative approach to performance modeling can in most cases pinpoint the real performance bottleneck, while also verifying the actual performance improvements due to various code optimizations. Moreover, the optimization techniques and performance modeling approach can both be applied to other convolution-intensive numerical methods for solving fractional differential equations.

*MSC 2010*: Primary 65Y05; Secondary 65L05, 65R20

*Key Words and Phrases*: fractional differential equation, fractional Adams-Bashforth-Moulton method, performance modeling, parallel computing, multicore CPU

## 1. Introduction

The time usage of a numerical program on a modern computer is often determined by how quickly data is moved within the entire memory system,

pp. xxx–xxx

rather than the speed of floating-point operations. This is no exception for the numerical schemes that solve fractional differential equations. While standard approaches to performance modeling concentrate on the amount of data that is moved in and out of the main memory, a closer look at the entire data flow can more precisely explain the actual performance obtained. On a modern CPU, we should thus direct our attention to the four connections of data flow: main memory$\leftrightarrow$L3 cache$\leftrightarrow$L2 cache$\leftrightarrow$L1 cache$\leftrightarrow$registers. While the bandwidth is increasing (or non-decreasing) along the four connections, the volume of data flow is also non-decreasing. Therefore, it is often impossible to say without analysis which connection may be the actual performance bottleneck.

This paper will use the fractional Adams-Bashforth-Moulton method [6, 5], referred to as the fractional Adams method later, as a representative numerical scheme that heavily relies on convolution calculations for solving fractional differential equations. As an extension of our earlier work [16], this paper reveals more details about code optimization in the context of a scalar equation. More importantly, we aim to propose a novel performance modeling approach, which can easily and accurately quantify the volumes of data flow through the four connections, with regard to different implementations of the fractional Adams method. In this way, the effect of various code optimizations, including parallelization, can be better understood.

## 2. Fractional Adams method

### 2.1. Target problem

Before showing the numerical algorithm of the fractional Adams method, let us first present this paper's target mathematical model, which is the following general initial-value problem with fractional differential order $\alpha > 0$:

$$D_*^\alpha y(t) = f(t, y(t)), \qquad y^{(k)}(0) = y_0^{(k)} \quad \text{for } k = 0, 1, \ldots, \lceil\alpha\rceil - 1, \tag{2.1}$$

where $t$ denotes time, $\lceil\cdot\rceil$ denotes the ceiling function, and $D_*^\alpha$ denotes a fractional differential operator that is defined in the sense of Caputo [2]:

$$D_*^\alpha y(t) := J^{\lceil\alpha\rceil-\alpha} D^{\lceil\alpha\rceil} y(t). \tag{2.2}$$

Here, $D^{\lceil\alpha\rceil}$ is the classical differential operator of integer order $\lceil\alpha\rceil$, and $J^\mu$ denotes the Riemann-Liouville integral operator of order $\mu > 0$, i.e.,

$$J^\mu y(t) := \frac{1}{\Gamma(\mu)} \int_0^t \frac{y(\tau)}{(t-\tau)^{1-\mu}} \,\mathrm{d}\tau, \tag{2.3}$$

where $\Gamma$ denotes Euler's Gamma function.

### 2.2. Numerical algorithm

The following is a brief description of the fractional Adams method, which finds the approximate solution of (2.1) on discrete time levels: $t_j = jh$, $j = 1, 2, \dots, N$, where $h$ is the time step size. More details can be found in [6, 5].

Let us denote by $y_j$ the numerical solution of $y(t_j)$. To find $y_{j+1}$, all the preceding values $y_0, y_1, \dots, y_j$ will be used, and the work consists of two sub-steps per time step. More specifically, a *predictor* first computes an intermediate approximation:

$$y^P_{j+1} = \sum_{k=0}^{\lceil\alpha\rceil-1} \frac{t^k_{j+1}}{k!} y^{(k)}_0 + h^\alpha \sum_{k=0}^{j} b_{j-k} f(t_k, y_k), \tag{2.4}$$

where the weights $b_i$ are defined by

$$b_i = \frac{(i+1)^\alpha - i^\alpha}{\Gamma(\alpha+1)}. \tag{2.5}$$

Then, a *corrector* uses the predicted value $y^P_{j+1}$ and computes $y_{j+1}$ by

$$y_{j+1} = \sum_{k=0}^{\lceil\alpha\rceil-1} \frac{t^k_{j+1}}{k!} y^{(k)}_0 + h^\alpha \left( c_j f(t_0, y_0) + \sum_{k=1}^{j} a_{j-k} f(t_k, y_k) + \frac{f(t_{j+1}, y^P_{j+1})}{\Gamma(\alpha+2)} \right), \tag{2.6}$$

where

$$a_i = \frac{(i+2)^{\alpha+1} - 2(i+1)^{\alpha+1} + i^{\alpha+1}}{\Gamma(\alpha+2)}, \quad c_j = \frac{j^{\alpha+1} - (j-\alpha)(j+1)^\alpha}{\Gamma(\alpha+2)}. \tag{2.7}$$

Some quick comments are in order here. First, the computations of time step $j+1$ are essentially two discrete convolutions between values of $f(t_k, y_k)$ and, respectively, the $a_{j-k}$ and $b_{j-k}$ weights. Second, the $a_i$ $(0 \le i \le N-1)$ and $b_i$ $(0 \le i \le N)$ weights should be pre-computed once and for all, to avoid repeatedly evaluating the costly power and Gamma functions. Third, the computational complexity for finding $y_1, y_2, \dots, y_N$ is $\mathcal{O}(N^2)$ floating-point operations, 50% multiplications and 50% additions. They are associated with the two increasingly longer convolutions per time step. The number of all other operations is $\mathcal{O}(N)$, thus of negligible cost.

## 3. High-performance implementations

The computational complexity and associated data intensity, shown above, call for high-performance computing. Our philosophy is that efficient serial implementations of the fractional Adams method lay the foundation of high-performance parallelizations. This section will therefore be divided into two parts, one about various serial implementations, the other about a resulting parallel implementation that uses MPI [8] programming.

### 3.1. Serial implementations

We have already touched upon the topic of serial and parallel programming in [16], which however addresses the different case of solving a system of fractional differential equations. Therefore, we will show in the following text the important details of coding serial implementations of the fractional Adams method for our scalar model problem (2.1).

**3.1.1.** *Common considerations.* As pointed out in Section **2**, the weights of $a_i$ and $b_i$ should be pre-computed before the time loop. These values can be stored in two one-dimensional arrays `a` and `b`. Another important observation is that the $f(t_k, y_k)$ values are also repeatedly used, thus needing another one-dimensional array, named `fy`. Note that the value of `fy[j+1]` is computed at the end of time step $j+1$, after $y_{j+1}$ is found by (2.4)-(2.7). A fourth one-dimensional array, named `y`, is assumed to store the numerical solutions of $y(t)$ at the discrete time levels. This array, however, is not as frequently used as `a`, `b` and `fy`.

**3.1.2.** *Baseline implementation.* Our first implementation naively follows the definition of the fractional Adams method given by (2.4)-(2.7). Skipping the data storage allocation and some initial calculations, we only show the main time loop as follows:

```
for (j=0; j<N; j++) {
  double prefix = y0+(j+1)*h*dy0+...  /* initial condition(s) */

  double sum_b = 0.0;
  for (k=0; k<=j; k++)
    sum_b += b[j-k]*fy[k];
  double yp_jp1 = prefix + h_alpha*sum_b; /* predictor value */

  double c_j = (pow(j,alpha+1)-(j-alpha)*pow(j+1,alpha))/gamma2;
  double sum_a = c_j*fy[0];
  for (k=1; k<=j; k++)
    sum_a += a[j-k]*fy[k];
  y[j+1] = prefix + h_alpha*(sum_a + f((j+1)*h, yp_jp1)/gamma2);
  fy[j+1] = f((j+1)*h, y[j+1]);
}
```

In the above code segment, `f` is an implementation of the right-hand function $f(t, y(t))$ of the model problem (2.1). Two constant variables have been pre-calculated as $\texttt{h\_alpha} = h^\alpha$ and $\texttt{gamma2} = \Gamma(\alpha + 2)$.

**3.1.3.** *Improvement by loop fusion.* During each `j`-iteration of the above baseline implementation, there are two for-loops both indexed with `k`. The `fy` array is thus traversed twice. There are two consequences of this approach. First, two values (`b[j-k]` and `fy[k]` or `a[j-k]` and `fy[k]`) need to be fed, per `k`-iteration, into the register file of a computer. Second, there is a high possibility that the second `k`-indexed `for`-loop has to reload the same `fy` values into certain levels of a computer's data cache, before feeding these values into the registers.

To counteract these two inefficiencies, the technique of *loop fusion* [7] can be applied. That is, the two `k`-indexed `for`-loops are merged together:

```
for (j=0; j<N; j++) {
  double prefix = y0+(j+1)*h*dy0+...

  double sum_b = b[j]*fy[0];
  double c_j = (pow(j,alpha+1)-(j-alpha)*pow(j+1,alpha))/gamma2;
  double sum_a = c_j*fy[0];

  for (k=1; k<=j; k++) {
    sum_b += b[j-k]*fy[k];
    sum_a += a[j-k]*fy[k];
  }

  double yp_jp1 = prefix + h_alpha*sum_b;
  y[j+1] = prefix + h_alpha*(sum_a + f((j+1)*h, yp_jp1)/gamma2);
  fy[j+1] = f((j+1)*h, y[j+1]);
}
```

It can be seen that this implementation avoids traversing the `fy` array twice during each `j`-iteration. The number of data values fed into the registers is thus reduced by 25%, in comparison with the baseline implementation. Moreover, we have halved the overhead that is associated with starting and stopping all the `k`-indexed `for`-loops.

**3.1.4.** *Improvement by loop unrolling.* To further decrease the total number of data values fed into the registers, the outer `j`-indexed time loop of the preceding loop-fusion version can be unrolled [7] with depth 2 as follows:

```
for (j=0; j<N; j+=2) {
  double prefix = y0+(j+1)*h*dy0+...
  double prefix1 = y0+(j+2)*h*dy0+...

  double sum_b = b[j]*fy[0];
  double c_j = (pow(j,alpha+1)-(j-alpha)*pow(j+1,alpha))/gamma2;
  double sum_a = c_j*fy[0];
```

```
    double sum_b1 = b[j+1]*fy[0];
    double c_jp1=(pow(j+1,alpha+1)-(j+1-alpha)*pow(j+2,alpha))/gamma2;
    double sum_a1 = c_jp1*fy[0];

    for (k=1; k<=j; k++) {
      sum_b += b[j-k]*fy[k];
      sum_a += a[j-k]*fy[k];
      sum_b1 += b[j+1-k]*fy[k];
      sum_a1 += a[j+1-k]*fy[k];
    }

    double yp_jp1 = prefix + h_alpha*sum_b;
    y[j+1] = prefix + h_alpha*(sum_a + f((j+1)*h, yp_jp1)/gamma2);
    fy[j+1] = f((j+1)*h, y[j+1]);

    double yp_jp2 = prefix1 + h_alpha*(sum_b1 + b[0]*fy[j+1]);
    y[j+2] = prefix1 + h_alpha*(sum_a1 + a[0]*fy[j+1]
                               + f((j+2)*h, yp_jp2)/gamma2);
    fy[j+2] = f((j+2)*h, y[j+2]);
  }
```

Now, the `k`-indexed inner `for`-loop is shared between the calculations of `y[j+1]` and `y[j+2]`. Consequently, `fy[k]` is used four times per `k`-iteration. The total amount of data fed into the registers is $\frac{5}{8} = 62.5\%$ of that of the baseline implementation.

**3.1.5.** *Improvement by alternating loop traversal.* In addition to feeding fewer data values into the registers, consideration can also be given to data reuse in the different cache levels, which typically use the LRU (*least recently used*) policy. More specifically, to improve data reuse of the `a`, `b` and `fy` arrays in the caches, we can alternate the traversal direction of two consecutive `k`-indexed `for`-loops. This alternation can enhance the performance of both the loop-fusion and loop-unroll versions. To save space, we only show the enhanced loop-fusion version as follows:

```
  for (j=0; j<N; ) {
    double prefix = y0+(j+1)*h*dy0+...

    double sum_b = b[j]*fy[0];
    double c_j = (pow(j,alpha+1)-(j-alpha)*pow(j+1,alpha))/gamma2;
    double sum_a = c_j*fy[0];

    for (k=1; k<=j; k++) {
      sum_b += b[j-k]*fy[k];
      sum_a += a[j-k]*fy[k];
    }
    double yp_jp1 = prefix + h_alpha*sum_b;
    y[j+1] = prefix + h_alpha*(sum_a + f((j+1)*h, yp_jp1)/gamma2);
    fy[j+1] = f((j+1)*h, y[j+1]);

    j++;
    prefix = y0+(j+1)*h*dy0+...

    sum_b = sum_a = 0.;
    for (k=j; k>=1; k--) {
      sum_b += b[j-k]*fy[k];
      sum_a += a[j-k]*fy[k];
    }

    sum_b += b[j]*fy[0];
    c_j = (pow(j,alpha+1)-(j-alpha)*pow(j+1,alpha))/gamma2;
    sum_a += c_j*fy[0];

    yp_jp1 = prefix + h_alpha*sum_b;
    y[j+1] = prefix + h_alpha*(sum_a + f((j+1)*h, yp_jp1)/gamma2);
    fy[j+1] = f((j+1)*h, y[j+1]);

    j++;
  }
```

We can see that, in comparison with the loop-fusion version from Section **3.1.3**, the number of $j$-iterations is halved. Now, each $j$-iteration contains two `k`-indexed `for`-loops, where the upper one iterates in the increasing order of `k`, and the lower one iterates oppositely.

**3.1.6.** *Some remarks about serial programming.* The four serial implementations have exactly the same amount of floating-point operations, but differ in how data are fed into a computer's registers and how much data reuse happens in the caches. These implementations are not meant to be conclusive with respect to code efficiency. One possibility is to unroll the outer `j`-indexed time loop deeper. However, caution must be exercised because deeper unrolls will require more registers. On a CPU that has a small number of registers, excessive unrolling will have negative effects on the performance due to register spilling [7]. Another possibility is to also unroll the `k`-indexed inner `for`-loop, for the purpose of using the vector computing units on modern CPUs. In this paper, we leave code vectorization to compilers, instead of hand-coding.

### 3.2. MPI-enabled parallel implementation

In [5], which was a ground-breaking paper on parallel computing and fractional calculus, parallelism was extracted from the j-indexed time loop. Core 1 of a multicore CPU was assigned to compute y[j] value, core 2 was assigned to y[j+1], and so on. While most of the computation could indeed be carried out concurrently among the CPU cores, there was a small segment of computation that had to be serialized. For example, cores 2,3,... had to wait for core 1 to finish first. Then, cores 3,4,... had to wait for core 2 to complete, and so on.

Our parallelization differs from [5] in that parallelism is now exploited with the k-indexed inner for-loop. (The time loop is still executed serially.) More specifically, each CPU core is assigned to compute one section of the k-loop. Then, all the CPU cores *collectively* carry out a parallel reduction operation that sums up the local results. In the context of MPI programming [8], each CPU core runs an MPI process that has a unique ID as my_rank. Then, every MPI process can individually calculate the following two integer variables:

```
int my_k_start = (my_rank*j)/num_procs+1;
int my_k_stop = ((my_rank+1)*j)/num_procs;
for (k=my_k_start; k<=my_k_stop; k++) {
  /* same code as before */
}
```

These two integers mark a region of the k-loop that is to be traversed only on the MPI process with my_rank as its ID. Afterward, a collective call to the MPI_Allreduce function, among all the MPI processes, can add up all the local sum_b and sum_a values. (The same work division is also applicable when reversing the traversal direction of the k-loop.) For the implementation based on loop fusion, one single call to MPI_Allreduce, which handles sum_a and sum_b together, is sufficient per time step (j-iteration). For the implementation based on loop unrolling, one single call to MPI_Allreduce is also sufficient per j-iteration, because sum_a, sum_b, sum_a1 and sum_b1 can be handled together.

### 3.3. OpenMP-enabled parallel implementation

Another simpler parallelization approach is to use OpenMP programming [3]. It suffices to insert a compiler directive before the k-loop, which also automatically carries out the needed parallel reduction operation:

```
#pragma omp parallel for reduction(+:sum_b,sum_a)
for (k=1; k<=j; k++) {
  /* same code as before */
}
```

For work division, the default static scheduler of OpenMP divides the k-iterations into $P$ chunks, where $P$ denotes the number of OpenMP threads. Thread 0 works on the first chunk (i.e., with the smallest k indices), thread 1 works on the second chunk, as so on. One potential problem, however, may arise in connection with reversing the loop traversal, i.e.,

```
#pragma omp parallel for reduction(+:sum_b,sum_a)
for (k=j; k>=1; k--) {
  /* ... */
}
```

Now, thread 0 is assigned to work with the largest k indices, and the other threads are assigned with k indices in a decreasing order. Although load balance is still perfect, the problem is that each thread has to switch between two regions of the k-iterations, possibly destroying temporal data locality in the caches and the NUMA architecture of memory.

Therefore, to get the best performance, MPI parallelization should be applied to the loop-unroll version with alternating loop traversal, whereas OpenMP parallelization should be applied to the plain loop-unroll version. The MPI-enabled parallel version thus runs faster than the OpenMP counterpart. However, one disadvantage with MPI parallelization, in addition to more programming effort, is that arrays a, b, y and fy are duplicated on every MPI process.

## 4. Performance modeling

The actual performance benefits from the various improvements of the baseline serial implementation depend on several factors. The main factor is the relationship between a CPU core's floating-point capability and its ability of moving required data through its entire memory system, all the way from the main memory to the registers. We will show in this section that it is possible to carry out detailed performance modeling of the various implementations of the fractional Adams method. This will pinpoint the performance bottlenecks of both serial and parallel implementations, thus providing a trustworthy understanding about the achievable performance.

In comparison with the famous *roofline* model [15], which focuses on a CPU's peak floating-point capability and its peak main memory bandwidth, our approach has two extensions. First, instead of only checking data movement between the main memory and the last-level cache, we also investigate possible bottlenecks in other sections of the data channel. Second, we adopt simple benchmark tests to obtain realistic bandwidth limits along the entire data channel, instead of only relying on the peak bandwidth of the main memory. Both extensions ensure a more accurate model.

### 4.1. Basic assumptions

For running the serial implementations, we assume a single CPU core that has three levels of data cache (L1,L2,L3) in addition to the main memory. The CPU core's floating-point capability (Gflop/s) is denoted by $F$. We use $B^r_{L1}$ to denote the bandwidth (GB/s) of reading data from L1 into the registers. Similarly, $B^r_{L2} \geq B^r_{L3} \geq B^r_M$ denote the L2→L1, L3→L2 and memory→L3 bandwidths, respectively. The reason for focusing only on the bandwidth of loading data is due to the infrequency of storing data in all the implementations.

Our fundamental assumption is that, due to modern CPUs' ability of pipelining and data prefetching [7], different actions can happen simultaneously. For example, while the floating-point units are doing calculations, the various data caches can be simultaneously busy with loading various data items that will be needed by subsequent calculations. Therefore, the lower bound of time usage of any serial implementation of the fractional Adams method can be described by the following formula:

$$\text{Time usage} = \max\left(\frac{C}{F}, \frac{D^r_{L1}}{B^r_{L1}}, \frac{D^r_{L2}}{B^r_{L2}}, \frac{D^r_{L3}}{B^r_{L3}}, \frac{D^r_M}{B^r_M}\right), \tag{4.8}$$

where $C$ denotes the amount of floating-point operations needed, $D^r_{L1}$ denotes the amount of data to be loaded from L1 to the registers, and so on. Here, we remark that (4.8) is based on the principle of *bound and bottleneck* [11], which was also adopted by the roofline model. The difference is that we also investigate possible bottlenecks due to loading data from the three levels of data cache. In the following text, our aim is to derive for each implementation the actual sizes of $C$, $D^r_{L1}$, $D^r_{L2}$, $D^r_{L3}$ and $D^r_M$, as functions of the problem size $N$ (the number of discrete time levels). The focus will be on the time usage due to operations associated with the `k`-indexed inner `for`-loops, i.e., of order $\mathcal{O}(N^2)$. Time usage of other operations, of order $\mathcal{O}(N)$, will be ignored. The latter includes the pre-calculation of $a_i$ and $b_i$ weights, and the associated initial cost of populating the different data caches.

### 4.2. The versions of baseline, loop-fusion and loop-unrolling

Independent of a particular serial implementation, we can "dissect" the entire problem size $N$ into four segments: $N_1 < N_2 < N_3 < N$, where

$$N_1 = \frac{\text{size of L1}}{3\times 8 \text{ bytes}}, \quad N_2 = \frac{\text{size of L2}}{3\times 8 \text{ bytes}}, \quad N_3 = \frac{\text{size of L3}}{3\times 8 \text{ bytes}}. \tag{4.9}$$

The $N_1$, $N_2$ and $N_3$ values are three sub-problem sizes that can be completely contained, respectively, in the three levels of data cache. That is, we have $D^r_M = 0$ for $j \leq N_3$, $D^r_{L3} = 0$ for $j \leq N_2$, and $D^r_{L2} = 0$ for $j \leq N_1$.

The reason for having $3 \times 8$ bytes in the denominators of (4.9) is due to the need of storing the required `a[j-k]`, `b[j-k]` and `fy[k]` values in caches, all in double precision (each value requires 8 bytes).

While the number of needed additions and multiplications for time step $j$ is $C = 2j + 2j = 4j$ (independent of implementation), the per-time-step values of $D^r_{L1}$, $D^r_{L2}$, $D^r_{L3}$ and $D^r_M$ depend on the particular implementation. More specifically, we have (assuming $j > N_3$)

$$\begin{aligned}
\text{Baseline version: } & D^r_{L1} = D^r_{L2} = D^r_{L3} = D^r_M = 4j \times 8\,\text{bytes},\\
\text{Loop-fusion version: } & D^r_{L1} = D^r_{L2} = D^r_{L3} = D^r_M = 3j \times 8\,\text{bytes},\\
\text{Loop-unroll version: } & D^r_{L1} = \frac{5j}{2} \times 8,\ \ D^r_{L2} = D^r_{L3} = D^r_M = \frac{3j}{2} \times 8\,\text{bytes}.
\end{aligned}$$

Consequently, the total time usage of the loop-unroll version (depth 2, see Section **3.1.4**) for $N$ time steps is

$$\begin{aligned}
\frac{N_1(N_1+1)}{2}\max\left(\frac{4}{F},\frac{20}{B^r_{L1}}\right) &+ \frac{(N_2-N_1)(N_2+N_1+1)}{2}\max\left(\frac{4}{F},\frac{20}{B^r_{L1}},\frac{12}{B^r_{L2}}\right)\\
&+\frac{(N_3-N_2)(N_3+N_2+1)}{2}\max\left(\frac{4}{F},\frac{20}{B^r_{L1}},\frac{12}{B^r_{L3}}\right)\\
&+\frac{(N-N_3)(N+N_3+1)}{2}\max\left(\frac{4}{F},\frac{20}{B^r_{L1}},\frac{12}{B^r_{M}}\right).
\end{aligned}$$

We note that the above formula of time usage has considered the fact of $B^r_{L1} \geq B^r_{L2} \geq B^r_{L3} \geq B^r_M$, which applies to all multicore CPUs. The time usage formulas for the baseline and loop-fusion versions are simpler.

### 4.3. The effect of alternating loop traversal

One common shortcoming with the above three serial implementations is that once the time step index $j$ exceeds a threshold value $N_i$, the data in the entire L$_i$ cache have to be reloaded from the L$_{i+1}$ cache for the next time step. This is due to the LRU caching policy. As we have already mentioned in Section **3.1.5**, the strategy of alternating the traversal direction of consecutive `k`-loops aims to improve data reuse in the caches.

More specifically, when $j$ exceeds $N_1$, alternating the `k`-loop traversal direction allows reusing, per time step, exactly $N_1$ values each of `a`,`b`,`fy` in L1. A similar benefit applies to L2 or L3, when $j$ exceeds $N_2$ or $N_3$. For instance, the cost of doing time steps $j = N_3+1, N_3+2, \ldots, N$ by the alternating-traversal enhanced loop-unroll implementation is

$$\max\left(\frac{(N-N_3)(N+N_3+1)}{2}\frac{4}{F}, \frac{(N-N_3)(N+N_3+1)}{2}\frac{20}{B^r_{L1}},\right.$$
$$\frac{(N-N_3)(N+N_3-2N_1+1)}{2}\frac{12}{B^r_{L2}}, \frac{(N-N_3)(N+N_3-2N_2+1)}{2}\frac{12}{B^r_{L3}},$$
$$\left.\frac{(N-N_3)(N-N_3+1)}{2}\frac{12}{B^r_M}\right).$$

### 4.4. Modeling the parallel implementation

To model the time usage by the parallelized implementations of Sections **3.2** and **3.3**, we can still use the formulas derived above. However, the definitions of $N_1$, $N_2$ and $N_3$ have to be modified, more specifically,

$$N_i = \frac{\text{number of used } \mathrm{L}_i \text{ caches} \times \text{size of } \mathrm{L}_i}{3 \times 8\,\text{bytes}}.$$

Here, we remark that the number of used L1 caches normally equals the number of MPI processes or OpenMP threads, because each CPU core has its own private L1 cache. The numbers of used L2 or L3 caches can be smaller than the number of used CPU cores, because of different degrees of L2/L3 cache sharing. Communication overhead, associated with the `MPI_Allreduce` function or OpenMP's intrinsic operations, is not considered by our performance model.

One subtle advantage of parallelization is that the aggregate $B^r_{L1}$, $B^r_{L2}$, $B^r_{L3}$, $B^r_M$ bandwidths that are accessible by multiple CPU cores are higher than those accessible by a single CPU core, although the ratio of increase is typically not proportional to the number of CPU cores used. As will be seen in the following section, we adopt a parallelized bandwidth benchmark to measure the aggregate values of $B^r_{L1}$,$B^r_{L2}$,$B^r_{L3}$,$B^r_M$ in the parallel setting.

## 5. Numerical experiment and time measurements

We adopt the same numerical example as in [5], namely,

$$D^{\alpha}_{*}y(t) = -y(t), \quad t \in (0,5],$$

where $\alpha = 1.3$ and the two initial conditions are $y(0) = 1$, $y'(0) = 0$. All the computations have been done using double precision, with numerical solutions verified against the analytical solution, which is the Mittag-Leffler function $y(t) = E_\alpha(-t^\alpha)$ [4].

### 5.1. Hardware platforms

The following three hardware systems that are based on multicore CPUs were used for our numerical experiment.

**System 1:** Two quad-core 2.0GHz Xeon E5504 *Nehalem-EP* CPUs [10]. Each core has its private L1 cache (32KB) and L2 cache (256KB). Four cores share an L3 cache (4MB). The used GNU C compiler has version 4.4.3 with optimization flag `-O2`.

**System 2:** Two 8-core 2.6GHz Xeon E5-2670 *Sandy Bridge-EP* C-PUs [9]. Each core has its private L1 cache (32KB) and L2 cache (256KB). Eight cores share an L3 cache (20MB). The used Intel C compiler has version 12.1.4 with optimization flag `-O2`.

**System 3:** Two 16-core 2.3GHz AMD Opteron 6276 *Interlagos* C-PUs. Each core has a private L1 cache (16KB). Two cores share an L2 cache (2MB), and four cores share an L3 cache (8MB) of which 6MB is used for caching data [1]. The used Cray C compiler has version 8.1.8 with optimization flag `-O2`.

### 5.2. Verification of performance model

The correctness of our overall performance model (4.8) depends on the predicted volumes of the data loads: $D^r_{L1}$, $D^r_{L2}$, $D^r_{L3}$ and $D^r_M$, in addition to the accuracy of the bandwidths. To verify the predicted values of $D^r_{L1}$ and $D^r_{L2}$, we have used the PAPI tool [12] to count two hardware events: PAPI_L1_DCA and PAPI_L2_DCA. These correspond to the total numbers of data accesses to the L1 and L2 caches.

In Table 1, the actual counts, which were associated with using three different C compilers on an AMD Opteron 6276 Interlagos CPU, are compared against our detailed performance models as described in Sections **4.2** and **4.3**. As we can see from the table, the accuracy of our predictions of $D^r_{L2}$ is confirmed. Measurements of PAPI_L1_DCA differ somewhat between the three compilers, where the GNU compiler matches with the predictions very well. One possible explanation is that the native Cray compiler may have achieved some level of register reuse, which is not considered by our prediction of $D^r_{L1}$. On the other hand, the Intel compiler may incur some additional data traffic from L1 cache to registers. Moreover, we have used the PAPI_FP_OPS counter of PAPI to confirm that the actual number of floating-point operations indeed remains as $2N^2$, for all the different implementations and compilers.

We can also mention that the same experiments were repeated by applying the three C compilers without optimization, i.e., with compilation option `-O0`. The measurements of PAPI_L2_DCA and PAPI_FP_OPS from this set of non-optimizing-compiler experiments are identical with those in

Table 1. Measurements of three PAPI events in comparison with predictions, associated with $N = 4 \times 10^5$ and four serial implementations. Three C compilers (Cray, GNU, Intel) were tested on an AMD Opteron 6276 Interlagos CPU core.

| Baseline serial implementation | | | | |
|---|---|---|---|---|
| PAPI event | Cray `cc -O2` | GNU `gcc -O2` | Intel `icc -O2` | Prediction |
| L1_DCA | $3.11 \times 10^{11}$ | $3.32 \times 10^{11}$ | $3.85 \times 10^{11}$ | $3.20 \times 10^{11}$ |
| L2_DCA | $4.14 \times 10^{10}$ | $4.01 \times 10^{10}$ | $4.03 \times 10^{10}$ | $4.00 \times 10^{10}$ |
| FP_OPS | $3.20 \times 10^{11}$ | $3.20 \times 10^{11}$ | $3.20 \times 10^{11}$ | $3.20 \times 10^{11}$ |
| Loop-fusion serial implementation | | | | |
| PAPI event | Cray `cc -O2` | GNU `gcc -O2` | Intel `icc -O2` | Prediction |
| L1_DCA | $2.54 \times 10^{11}$ | $2.74 \times 10^{11}$ | $3.53 \times 10^{11}$ | $2.40 \times 10^{11}$ |
| L2_DCA | $3.18 \times 10^{10}$ | $3.04 \times 10^{10}$ | $3.05 \times 10^{10}$ | $3.00 \times 10^{10}$ |
| FP_OPS | $3.20 \times 10^{11}$ | $3.20 \times 10^{11}$ | $3.20 \times 10^{11}$ | $3.20 \times 10^{11}$ |
| Loop-unroll serial implementation | | | | |
| PAPI event | Cray `cc -O2` | GNU `gcc -O2` | Intel `icc -O2` | Prediction |
| L1_DCA | $1.74 \times 10^{11}$ | $2.09 \times 10^{11}$ | $2.72 \times 10^{11}$ | $2.00 \times 10^{11}$ |
| L2_DCA | $1.56 \times 10^{10}$ | $1.51 \times 10^{10}$ | $1.52 \times 10^{10}$ | $1.50 \times 10^{10}$ |
| FP_OPS | $3.20 \times 10^{11}$ | $3.20 \times 10^{11}$ | $3.20 \times 10^{11}$ | $3.20 \times 10^{11}$ |
| Loop-unroll + alternating traversal serial implementation | | | | |
| PAPI event | Cray `cc -O2` | GNU `gcc -O2` | Intel `icc -O2` | Prediction |
| L1_DCA | $1.74 \times 10^{11}$ | $2.08 \times 10^{11}$ | $2.94 \times 10^{11}$ | $2.00 \times 10^{11}$ |
| L2_DCA | $1.57 \times 10^{10}$ | $1.51 \times 10^{10}$ | $1.52 \times 10^{10}$ | $1.50 \times 10^{10}$ |
| FP_OPS | $3.20 \times 10^{11}$ | $3.20 \times 10^{11}$ | $3.20 \times 10^{11}$ | $3.20 \times 10^{11}$ |

Table 1, which were produced by optimizing compilers. The measurements of PAPI_L1_DCA from the non-optimizing-compiler experiments are considerably higher, because of ineffective use of the registers and L1 cache due to no compiler optimization. In other words, our predictions of $D^r_{L1}$ assume effective use of the registers and L1 cache, thus compatible with compiler optimizations that are commonly used in practice.

## 5.3. Serial performance

Table 2 compares the actual time measurements (denoted by $T_A$) with the predicted time usages (denoted by $T_P$) that are produced by the performance models from Sections **4.2-4.3**. These are associated with running the four serial implementations on a single core of the three chosen hardware platforms. It can be seen that the three latter serial implementations progressively improve the performance of the baseline version. The $B^r_{L1}$,

$B^r_{L2}$, $B^r_{L3}$, $B^r_M$ bandwidths are measured by the dot-product benchmark (without manual loop unrolling) from STREAM2 [14]. The predicted time usages have a similar trend as that of the actual time measurements on all three platforms, as depicted in Figure 1.

TABLE 2. Actual time measurements ($T_A$) and predicted time usages ($T_P$) of four serial implementations.

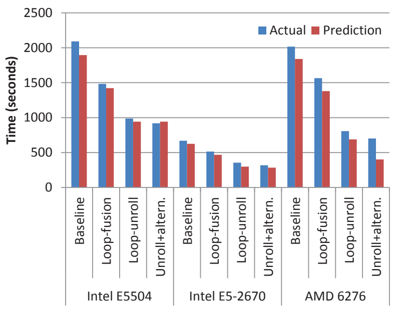
| One core of Nehalem-EP E5504 CPU | | | | | | |
|---|---|---|---|---|---|---|
| $B^r_{L1} = 10.61$GB/s, $B^r_{L2} = 10.58$GB/s, $B^r_{L3} = 10.53$GB/s, $B^r_M = 8.39$GB/s | | | | | | |
| | $N = 2 \times 10^5$ | | $N = 10^6$ | | $N = 4 \times 10^6$ | |
| | $T_A$ | $T_P$ | $T_A$ | $T_P$ | $T_A$ | $T_P$ |
| Baseline | 69.08 | 64.44 | 2091.69 | 1895.19 | 33727.51 | 30500.68 |
| Loop-fusion | 50.96 | 48.33 | 1482.64 | 1421.39 | 23931.38 | 22875.51 |
| Loop-unroll | 37.48 | 37.70 | 987.27 | 942.51 | 15936.27 | 15080.11 |
| Unroll+altern. | 34.12 | 37.70 | 919.94 | 942.51 | 14917.16 | 15080.11 |

| One core of Sandy Bridge-EP E5-2670 CPU | | | | | | |
|---|---|---|---|---|---|---|
| $B^r_{L1} = 35.31$GB/s, $B^r_{L2} = 35.14$GB/s, $B^r_{L3} = 30.22$GB/s, $B^r_M = 17.16$GB/s | | | | | | |
| | $N = 2 \times 10^5$ | | $N = 10^6$ | | $N = 4 \times 10^6$ | |
| | $T_A$ | $T_P$ | $T_A$ | $T_P$ | $T_A$ | $T_P$ |
| Baseline | 25.23 | 21.16 | 669.22 | 624.72 | 15471.11 | 14610.73 |
| Loop-fusion | 18.67 | 15.87 | 513.74 | 468.54 | 11268.31 | 10958.05 |
| Loop-unroll | 13.62 | 11.32 | 355.31 | 298.91 | 6793.97 | 5543.67 |
| Unroll+altern. | 12.54 | 11.32 | 317.72 | 283.21 | 6072.72 | 4531.29 |

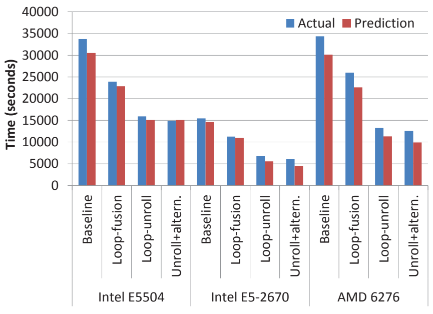
| One core of Interlagos 6276 CPU | | | | | | |
|---|---|---|---|---|---|---|
| $B^r_{L1} = 59.48$GB/s, $B^r_{L2} = 27.87$GB/s, $B^r_{L3} = 12.59$GB/s, $B^r_M = 8.48$GB/s | | | | | | |
| | $N = 2 \times 10^5$ | | $N = 10^6$ | | $N = 4 \times 10^6$ | |
| | $T_A$ | $T_P$ | $T_A$ | $T_P$ | $T_A$ | $T_P$ |
| Baseline | 51.23 | 45.51 | 2015.51 | 1839.14 | 34360.84 | 30141.03 |
| Loop-fusion | 44.38 | 34.13 | 1564.61 | 1379.35 | 25999.14 | 22605.77 |
| Loop-unroll | 25.43 | 17.06 | 806.91 | 689.68 | 13252.49 | 11302.88 |
| Unroll+altern. | 20.82 | 8.55 | 701.66 | 401.39 | 12587.08 | 9901.72 |

### 5.4. Parallel performance

Table 3 concerns the MPI-enabled parallelization of the loop-unroll version enhanced with alternating loop traversal, as described in Section **3.2**. The $B^r_{L1}$, $B^r_{L2}$, $B^r_{L3}$, $B^r_M$ bandwidths, now as functions of the number of MPI processes used, are measured by an MPI extension of the dot-product benchmark. Again, the trend of the actual time measurements follows that of the predicted time usages. The noticeable superlinear speedup results



$N = 10^6$



$N = 4 \times 10^6$

FIGURE 1. Comparing predicted and actual time usages of four serial implementations and two problem sizes.

from 1 core to 2 cores, obtained on the Interlagos system, are also justified by the predictions. This is because the measured $B^r_{L1}$, $B^r_{L2}$, $B^r_{L3}$, $B^r_M$ bandwidths almost perfectly double from the case of 1 core to the case of 2 cores, at the same time as the aggregate L1/L2/L3 cache sizes double. That is, using 2 cores has an inherent advantage over using 1 core. On the other hand, when the number of MPI processes gets closer to the maximum number of CPU cores, neither the aggregate $B^r_{L1}$,$B^r_{L2}$,$B^r_{L3}$,$B^r_M$ bandwidths nor the aggregate L1/L2/L3 cache sizes increase linearly. Sublinear speedup is thus the result, also predicted by the quantitative performance model.

One negative factor for speedup is the MPI overhead, which is not considered in our performance models. Actually, the calls to `MPI_Allreduce` can consume a considerable amount of time. For example, for the problem size of $N = 10^6$, the Scalasca tool [13] reveals that the percentage of MPI time usage on the Sandy-Bridge system arises from 1.6% for two cores to 20.8% for 16 cores.

We have measured the performance of an OpenMP-parallelization of the serial implementation of plain loop-unroll, see Section **3.3**. The trend of the actual/predicted OpenMP time usage is similar to that of MPI, as seen in Figure 2. As explained in Section **3.3**, the MPI parallelization is based on a better serial implementation, therefore the better parallel MPI performance. Tests of an OpenMP-parallelization of loop-unroll + alternating traversal (not shown in Figure 2) also confirm its inferior performance in comparison with the OpenMP-parallelization of plain loop-unroll.

TABLE 3. Actual time measurements ($T_A$) and predicted time usages ($T_P$) of an MPI parallelization of loop unroll+alternating traversal.

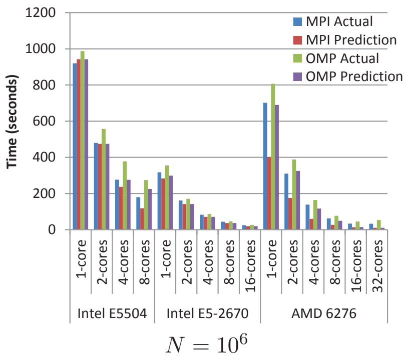
| Two quad-core Nehalem-EP E5504 CPUs | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | | | | $N = 10^6$ | | $N = 4 \times 10^6$ | |
| MPI | $B^r_{L1}$ | $B^r_{L2}$ | $B^r_{L3}$ | $B^r_M$ | $T_A$ | $T_P$ | $T_A$ | $T_P$ |
| 1 proc | 10.61 | 10.58 | 10.53 | 8.39 | 919.94 | 942.51 | 14917.16 | 15080.11 |
| 2 procs | 21.07 | 21.07 | 20.89 | 15.71 | 479.53 | 474.61 | 7920.81 | 7593.73 |
| 4 procs | 42.21 | 42.21 | 42.03 | 21.31 | 276.75 | 236.91 | 5036.66 | 3790.57 |
| 8 procs | 84.55 | 84.55 | 83.51 | 24.99 | 179.64 | 118.27 | 3672.05 | 3213.96 |
| Two 8-core Sandy Bridge-EP E5-2670 CPUs | | | | | | | | |
| | | | | | $N = 10^6$ | | $N = 4 \times 10^6$ | |
| MPI | $B^r_{L1}$ | $B^r_{L2}$ | $B^r_{L3}$ | $B^r_M$ | $T_A$ | $T_P$ | $T_A$ | $T_P$ |
| 1 proc | 35.31 | 35.14 | 30.22 | 17.16 | 317.72 | 283.21 | 6072.72 | 4531.29 |
| 2 procs | 70.47 | 70.26 | 60.44 | 36.25 | 161.64 | 141.91 | 2829.93 | 2270.47 |
| 4 procs | 141.21 | 140.45 | 119.44 | 64.92 | 82.65 | 70.81 | 1493.09 | 1133.06 |
| 8 procs | 272.86 | 270.75 | 229.89 | 88.86 | 43.78 | 36.64 | 857.77 | 586.38 |
| 16 procs | 513.63 | 511.03 | 426.83 | 87.71 | 24.87 | 19.46 | 632.92 | 406.51 |
| Two 16-core Interlagos 6276 CPUs | | | | | | | | |
| | | | | | $N = 10^6$ | | $N = 4 \times 10^6$ | |
| MPI | $B^r_{L1}$ | $B^r_{L2}$ | $B^r_{L3}$ | $B^r_M$ | $T_A$ | $T_P$ | $T_A$ | $T_P$ |
| 1 proc | 59.48 | 27.87 | 12.59 | 8.48 | 701.66 | 401.39 | 12587.08 | 9901.72 |
| 2 procs | 118.89 | 54.62 | 23.74 | 16.75 | 310.35 | 175.42 | 6133.58 | 4361.55 |
| 4 procs | 237.88 | 110.53 | 47.72 | 33.69 | 138.80 | 59.72 | 2862.12 | 1682.05 |
| 8 procs | 475.78 | 221.07 | 84.43 | 55.08 | 62.81 | 26.84 | 1430.09 | 978.43 |
| 16 procs | 875.64 | 427.36 | 134.22 | 66.72 | 33.92 | 13.73 | 755.04 | 635.81 |
| 32 procs | 946.48 | 741.69 | 162.38 | 65.81 | 32.67 | 10.56 | 740.04 | 637.87 |

## 6. Extension to solving a system of fractional equations

To demonstrate that our performance modeling approach can be extended to other situations, let us revisit the specific case of using the fractional Adams method for a system of four fractional equations, as discussed in [16]. That is, the target mathematical model now becomes

$$\begin{aligned} D_*^{1/2} y_1(t) &= y_2(t), \\ D_*^{1/2} y_2(t) &= y_3(t), \\ D_*^{1/2} y_3(t) &= y_4(t), \\ D_*^{1/2} y_4(t) &= A^{-1}(-Cy_1(t) - By_4(t) + f(t)), \end{aligned} \tag{6.10}$$

with appropriate initial conditions.



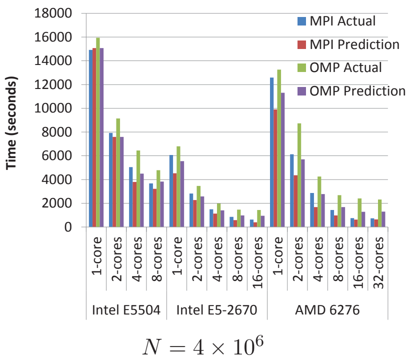


FIGURE 2. Comparing predicted and actual time usages of an MPI-parallelization and an OpenMP-parallelization on three parallel platforms, for two problem sizes.

In total, seven arrays are needed for implementing the fractional Adams solver for this case. In addition to `a` and `b`, arrays `y1`, `y2`, `y3` and `y4` will store the numerical solutions. Moreover, during the convolution calculations, to avoid unnecessary repetitions of computing the right-hand term of $A^{-1}(-Cy_1(t) - By_4(t) + f(t))$, we adopt another array named `g4yt`.

Due to space limit, we will only show the `k`-indexed inner `for`-loop that belongs to the version of loop-fusion for this four-equation solver:

```
for (k=1; k<=j; k++) {
  sum_b1 += b[j-k]*y2[k];
  sum_b2 += b[j-k]*y3[k];
  sum_b3 += b[j-k]*y4[k];
  sum_b4 += b[j-k]*g4yt[k];
  sum_a1 += a[j-k]*y2[k];
  sum_a2 += a[j-k]*y3[k];
  sum_a3 += a[j-k]*y4[k];
  sum_a4 += a[j-k]*g4yt[k];
}
```

It should be noticed that six arrays (excluding `y1`) are repeatedly used in the above convolution calculations. Consequently, we have $D^r_{L1} = D^r_{L2} = D^r_{L3} = D^r_M = 6j \times 8$ bytes for time step $j$. In comparison, for a baseline implementation, where there are two separate `k`-indexed `for`-loops per time step, we have $D^r_{L1} = D^r_{L2} = D^r_{L3} = D^r_M = 10j \times 8$ bytes. Without going into the details, we can state that loop unrolling and alternating loop traversal can help reducing the values of $D^r_{L1}$, $D^r_{L2}$, $D^r_{L3}$ and $D^r_M$. Another detail is that $N_1$, $N_2$ and $N_3$ are calculated by the same formulas as (4.9), but we need to replace the denominator with $6 \times 8$ bytes. This is because six arrays (instead of three) are now repeatedly used in the convolution loops.

## 7. Concluding remarks

The performance models that have been proposed are conceptually very simple. They are also easy to use, because the only required hardware-specific parameters are the different data cache sizes, the peak floating-point capability $F$, plus the realistic bandwidths that can be easily measured by running a (parallelized) dot-product benchmark. The accuracy of the predicted time usages can be affected by several factors. For example, a compiler may transform the code in a different way than expected. Moreover, the actual code may suffer from stalls that are not considered in the performance models. Neither are the data caches exclusively used for storing `a[j-k]`, `b[j-k]` and `fy[k]`. Inaccuracy may also arise from inaccurate measurements of the bandwidths. Nevertheless, the proposed performance models are able to identify the true bottlenecks in most cases, as well as verifying the performance advantage of various code improvements.

We have also demonstrated, through actual time measurements, the effect of various performance-enhancing strategies. Although the serial and parallel implementations are specific for the fractional Adams method, we believe that these are of value to similar numerical schemes, which are based on convolution calculations for the temporal integration, for solving fractional differential equations.

**Acknowledgement**

The first author is funded by a grant from the China Scholarship Council. A part of the computer time used for this paper was provided by NOTUR: the Norwegian Metacenter for Computational Science.

## References

[1] AMD Opteron$^{\text{TM}}$ 6200 Series Processors: Linux Tuning Guide, 2012.
[2] M. Caputo, Linear models of dissipation whose Q is almost frequency independent II. *Geophys. J. Int.*, **13**, No 5 (1967), 529-539.
[3] B. Chapman, G. Jost and R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming.* MIT Press, Cambridge, MA (2007).

The source codes used in this paper can be downloaded from `http://heim.ifi.uio.no/xingca/fractional_ABM.tar`.

[4] K. Diethelm, *The Analysis of Fractional Differential Equations.* Springer, Berlin (2010).

[5] K. Diethelm, An efficient parallel algorithm for the numerical solution of fractional differential equations. *Fract. Calc. Appl. Anal.* **14**, No 3 (2011), 475-490.

[6] K. Diethelm, N. J. Ford and A. D. Freed, A predictor-corrector approach for the numerical solution of fractional differential equations. *Nonlinear Dynamics* **29** (2002), 3-22.

[7] S. Goedecker and A. Hoisie, *Performance Optimization of Numerically Intensive Codes*. SIAM, Philadelphia (2001).

[8] W. Gropp and E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface.* MIT Press, Cambridge, MA (1994).

[9] Intel Xeon Processor E5-2670. http://ark.intel.com/products/64595/.

[10] Intel Xeon Processor E5504. http://ark.intel.com/products/40711/.

[11] E. D. Lazowska, J. Zahorjan, G. S. Graham and K. C. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models.* Prentice-Hall, Upper Saddle River, NJ (1984).

[12] PAPI: Performance Application Programming Interface. http://icl.cs.utk.edu/papi/.

[13] Scalasca homepage. http://www.scalasca.org/.

[14] The STREAM2 Home Page. http://www.cs.virginia.edu/stream/stream2/.

[15] S. Williams, A. Waterman and D. Patterson, Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* **52**, No 4 (2009), 65-76.

[16] W. Zhang and X. Cai, Efficient implementations of the Adams-Bashforth-Moulton method for solving fractional differential equations. In: *Proceedings of FDA'12*, Nanjing (2012).

[1] *Department of Informatics*
*University of Oslo*
*P.O. Box 1080 Blindern*
*N-0316 Oslo, NORWAY*

*e-mail: weizhang@student.matnat.uio.no*

[2] *Simula Research Laboratory*
*P.O. Box 134*
*N-1325 Lysaker, NORWAY*

*e-mail: {wenjie,xingca}@simula.no*

*Accepted: March, 2014*