Parallel Scaling Performance and Higher-Order Methods

A. Jared Buckley[1] **and B. Gaurav Khanna**[1]
[1]Physics Department, University of Massachusetts Dartmouth, Dartmouth, MA, USA

Abstract— *There is considerable current interest in higher-order methods and also large-scale parallel computing in nearly all areas of science and engineering. In this work, we take a number of basic finite-difference stencils that compute a numerical derivative to different orders of accuracy and carefully study the scaling performance of each, on a parallel computer cluster. We conclude that if one has a code that exhibits a high order of convergence, then there is likely to be no significant gain through cluster parallelism in the context of total execution or "wall clock" time. Conversely, for a low order code that exhibits good parallel scaling, there is insignificant gain through the implementation of a higher-order convergent algorithm.*

Keywords: higher-order, finite-difference, parallel, scaling

1. Introduction

In recent decades there has been a tremendous rise in numerical computer simulations, in nearly every area of science and engineering. This is largely due to the development of (Beowulf) cluster parallel computing that involves connecting together "off-the-shelf" computing units (for example, commodity desktop or laptop computer processors) into a configuration that would achieve the same level of performance, or even outperform, traditional supercomputers at a fraction of the cost [1]. The main reason behind the significant cost benefit of cluster computing is that it is entirely based on mass-produced, consumer hardware. Computational science has benefited and expanded tremendously in the last decade due to the rapid improvements in processor performance (*Moore's Law*) and major price drops due to mass production and associated market forces.

In addition to the strong increase in the interest in parallel cluster computing, there is also a rising trend in developing numerical algorithms that converge faster than the common second-order accurate schemes [2]. Some examples of such higher-order convergent methods are – higher-order finite-differencing, spectral collocation method, radial basis function method, finite-element and others [3], [4], [5], [6], [7], [8], [9], [10], [11]. In this work, we restrict ourselves to higher-order finite-difference schemes, however, we anticipate that our findings are generic enough that they would apply to any higher-order method.

The main goal of this work is to clearly demonstrate a form of "trade-off" between parallel computing and higher-order methods. This trade-off stems from the detailed parallel scaling behavior of the various higher-order schemes under specific conditions. In particular, our main assumption in this work is that the physical or engineering problem to be solved numerically has a known degree of tolerance or error acceptable for the solution, and is a given fixed quantity. We interpret this error to be the scale of the "discretization" or truncation error arising from the numerical scheme, which is, of course, a significant simplification – however, one that is reasonable for a wide class of problems. In other words, *we study the scaling performance of different order finite-difference methods given a fixed level of the discretization error*.

The outcome of our study suggests several very significant conclusions: (a) If one has a parallel code that scales well and exhibits second-order convergence, there is insignificant gain to be expected from a higher-order method implementation, assuming one has a large enough computational resource available, and the major consideration is the total execution time; (b) if one has a serial code exhibiting higher-order convergence (say, higher than fourth-order) then there is no significant gain from a parallel algorithm in a similar context; and (c) depending upon the acceptable error level, there is likely an optimal approach i.e. a combination of parallelism and method-order that would be ideal for the problem.

This article is organized as follows: In Section 2, we present a simple parallel scaling model that predicts the outcome of our planned study, based on simple heuristic reasoning. In Section 3, we detail the method of our study and present explicit mathematical expressions and our approach towards cluster parallelism. In Section 4, we show and discuss our results, and we end with some conclusive remarks in Section 5.

2. Simple Scaling Model

In this section, we present a simple model that predicts the parallel scaling behavior for a finite-difference method of any order. The model will help explain our findings and may provide some predictive value for other codes beyond the simple sample code we consider in this work.

Let us say that one is interested in performing a simple one-dimensional (1D) numerical derivative using a second-order finite-difference stencil. An important parameter that must be chosen is the grid resolution, that is typically set by the grid size N. The number of numerical calculations necessary to perform the derivative computation on the entire grid would then be on the order of N (more accurately, it would

be closer to $2N$ calculations – but let us ignore the constant pre-factors for this discussion). Now, let's assume that one attempts the same computation on a large parallel cluster with n processors using a standard domain-decomposition approach. Each processor would then perform N/n computations and would have to communicate two values (the boundaries of its subgrid) to neighboring processors. The parallel scaling behavior is largely determined by the ratio of the time-scale associated with this communication, to the time-scale of the actual numerical calculations on the subgrid. For good scaling, the entire computation should be heavily dominated by the calculations being performed by the processors and not by the communication. Therefore, as N increases, better scaling behavior is expected here.

Now, let us consider the same in the context of a higher-order finite-difference stencil of order p in 1D. For the *same* level of error as the second-order case above, one would only need a grid size on the scale of $N^{2/p}$. Thus, in a parallel computation environment, with n processors, each subgrid would be of size $\frac{N^{2/p}}{n}$ and the number of computations performed by each processor would be on the scale of $\frac{p}{n}N^{2/p}$. The number of values to communicate from one processor to another would increase to p. This is explained in detail in the next section. Thus, the ratio of computation to communication would behave as $N^{2/p}$ which drops dramatically as one increases the method order p, due to the power of $2/p$ in the expression[1]. This implies that *the scaling of higher-order methods, in general, is expected to be worse compared to the second-order case, in the context of a fixed discretization error.*

The question that we will address in this work, is whether the improved scaling of lower order methods is *enough* to give them an advantage in the context of the most important aspect of a numerical computation – the *total execution or "wall clock" time*. In fact, given our model above, we can make an estimate of how this could occur. Let us assume that for a given higher-order method, say, fourth-order and a problem size of interest, the parallel scaling performance is such that one is actually better off simply utilizing a serial code. The execution time would then be on the scale of $\sqrt{N}$. On the other hand, assuming that the second-order code has better parallel scaling, as would be expected, one would estimate the wall clock time to be on the scale of N/n. Thus, if one has computational resources with $n \sim \sqrt{N}$ at one's disposal, then at least in terms of total execution time, the two methods will complete the computation on the same time-scale. For the case of order p, this would change to $n \sim \frac{N^{1-2/p}}{p}$. This is the main point that we explicitly investigate in the following sections using finite-difference schemes of order 2, 4, 6 and 8.

[1]Since such communication is typically *latency* bound, as opposed to *bandwidth* bound, the p-dependence of this ratio is better estimated to be $pN^{2/p}$.

3. Methodology

In this section, we describe in detail the method of study adopted in this work. We begin with a discussion of higher-order finite-difference stencils, followed by our results from correlating the error with grid size N and end with a description of our parallel code implementation.

For the finite-difference calculations, we used a cosine function on a 1D domain from 0 to 12π:

$$f(x) = \cos x, \;\; x \in [0, 12\pi)$$

Now, for a numerical implementation, x is discretized simply as:

$$x_i = ih$$

where

$$h = \frac{12\pi}{N}$$

and i is an index that labels an arbitrary grid point on the domain.

To calculate the derivative of a function at grid point i using the finite-difference schemes, it is necessary to use the function values at neighboring grid points. As the order of the scheme increases, more grid points are needed for the calculation.

In the context of this work, we focus our attention on the first derivative of $f(x)$. At grid point i, using the various different order central finite-difference schemes, the derivative is given as [2]:
Order 2:

$$\frac{1}{h}(\frac{-1}{2}f_{i-1} + \frac{1}{2}f_{i+1})$$

Order 4:

$$\frac{1}{h}(\frac{1}{12}f_{i-2} + \frac{-2}{3}f_{i-1} + \frac{2}{3}f_{i+1} + \frac{-1}{12}f_{i+2})$$

Order 6:

$$\frac{1}{h}(\frac{-1}{60}f_{i-3} + \frac{3}{20}f_{i-2} + \frac{-3}{4}f_{i-1} + \frac{3}{4}f_{i+1} + \frac{-3}{20}f_{i+2} + \frac{1}{60}f_{i+3})$$

Order 8:

$$\frac{1}{h}(\frac{1}{280}f_{i-4} + \frac{-4}{105}f_{i-3} + \frac{1}{5}f_{i-2} + \frac{-4}{5}f_{i-1} + \frac{4}{5}f_{i+1} + \frac{-1}{5}f_{i+2} + \frac{4}{105}f_{i+3} + \frac{-1}{280}f_{i+4})$$

where the notation, $f_i = f(x_i)$.

It is clear from the above expressions that finite-difference schemes at higher orders produce an increasingly wider stencil. These wide stencils become important in the context of parallel computing as the passing of messages increases significantly with stencil size. As an example, a grid point

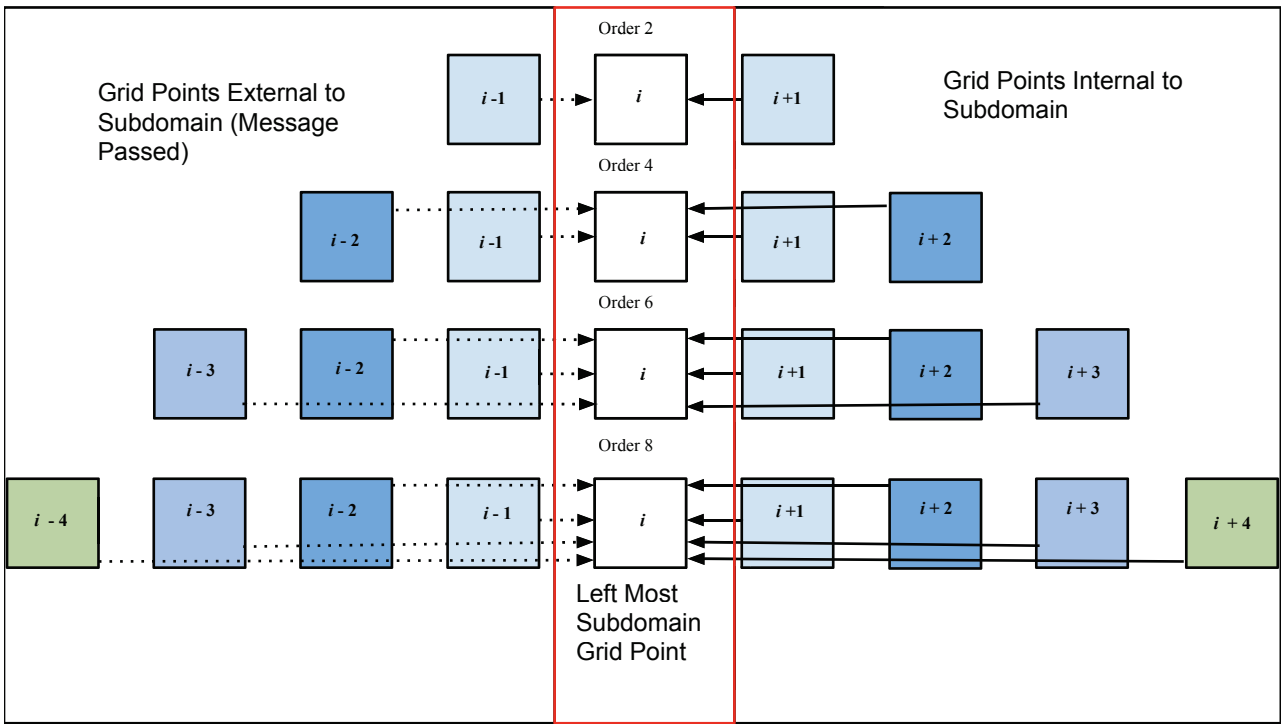

Fig. 1: The stencil structure for various finite-difference orders on a parallel computer cluster. The grid point labelled with index i is meant to reside on the left-edge of a subdomain. The grid points further on the left, must be communicated over the network (dotted-lines) while the ones on the right do not (solid-lines). Note that for higher-order stencils information from additional grid points have to be communicated across the network.

calculation at the edge of a process' subdomain would need 1 grid point value passed from outside the subdomain for order 2, while order 8 would require 4 grid point values to be passed. It is this communication overhead that influences the parallel scaling of the finite-difference schemes as described in Section 2. We developed computational routines, written in the C programming language, to better understand the behavior of the parallel scaling over a network.

Because we are concerned with the scaling at a fixed level of error, it was first necessary to correlate N with a given error level. This was achieved using an iterative algorithm that searched for a given error value for each finite-difference order being investigated. The algorithm calculated the first derivative of the cosine function in two ways: using the math library sine function and using the finite-difference formula at a value of N. The error was calculated at each point in the domain, and the maximum error on the domain was compared to the given error value. If the calculated error was less than the given error, the value of N was recorded; otherwise, N was incremented up by one and the process was repeated. The correlation of N with the error for each investigated finite-difference order is given in Tab. 1 and Fig. 2. The values of N fit the expected patterns extremely well.

With the correlation of N and error known, we developed a parallel message-passing (MPI) [12] routine to study the scaling behavior of the finite-difference schemes at fixed error values. The MPI routine divided the domain into even subdomains, with each subdomain associated with an MPI rank. Separate routines were developed for finite-difference orders 2, 4, 6, and 8. Each routine contained the appropriate finite-difference stencil and a modified MPI communication setup to allow for the transfer of the appropriate grid point values. MPI calls for higher-order cases were set to have higher buffer sizes to accommodate the increased need for grid points external to an MPI rank subdomain. Before any finite-difference calculations were performed, MPI ranks communicated in order to transfer their respective subdomain edges to the nearest logical MPI rank. We used MPI blocking calls for all communication. To prevent blocking calls from locking up the routine, communication was broken into two steps. The left edge of the MPI rank subdomains was sent to the nearest rank on the left (rank 0 excluded), then the right edge of the MPI rank subdomains was sent to the nearest rank on the right (maximum rank excluded). This approach is depicted graphically in Fig. 1. Once communication was completed, each rank independently computed the finite-difference first derivative of their respective subdomain. This process was repeated several times to allow for measurable execution wall clock times.

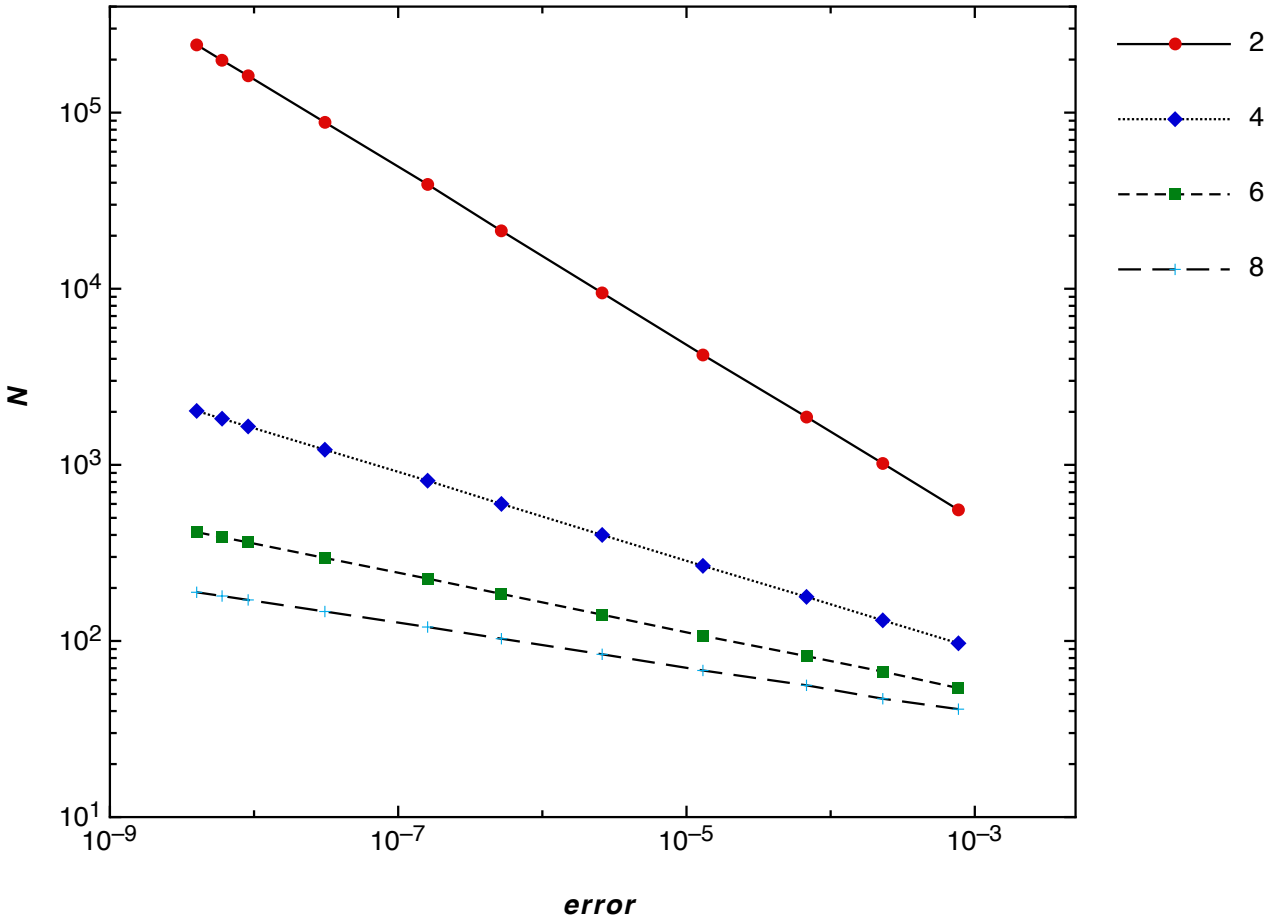

Fig. 2: The correlation of N with the error for various finite-difference orders. The powers of N computed from this data are: -1.998, -3.955, -5.942 and -7.876 respectively.

Our test cluster was Air Force Research Lab's CONDOR supercomputer. This system is a heterogeneous supercomputer comprised of commercial-off-the-shelf commodity components with 500 TFLOPS of processing power, and is a"green" supercomputer, designed to consume significantly less energy than comparable supercomputers [13]. All computations were performed using quadruple-precision floating-point numerical accuracy, as is often necessary in the context of higher-order methods, in order to reduce the roundoff errors to acceptable levels.

Table 1: Correlation of N and Error

Error	2	4	6	8
7.7×10^{-4}	555	97	54	41
2.3×10^{-4}	1019	131	67	47
6.8×10^{-5}	1871	178	82	56
1.3×10^{-5}	4210	267	107	68
2.6×10^{-6}	9473	400	141	84
5.2×10^{-7}	21313	600	185	103
1.6×10^{-7}	39153	813	226	120
3.1×10^{-8}	88094	1219	296	147
9.1×10^{-9}	161843	1652	363	171
6.0×10^{-9}	198222	1829	388	180
4.0×10^{-9}	242766	2024	415	189

4. Results

In this section we present the results obtained due to the approach and methodology as detailed in the previous sections.

In Fig. 3 we depict the speedup as a function of the number of processors n, for a grid size $N = 39,153$ or error-level 1.5×10^{-7}. The speedup is defined relative to the second-order code running on a single processor. We show the same for all the method orders considered in this work, namely 2, 4, 6 and 8. The expectations, as presented in Section 2, are clearly borne out in our speedup data. The second-order method, scales the best with n, and the higher-order cases exhibit much poorer scaling. In fact, the sixth and eight-order cases do not scale at all! One is clearly better off running those in serial mode.

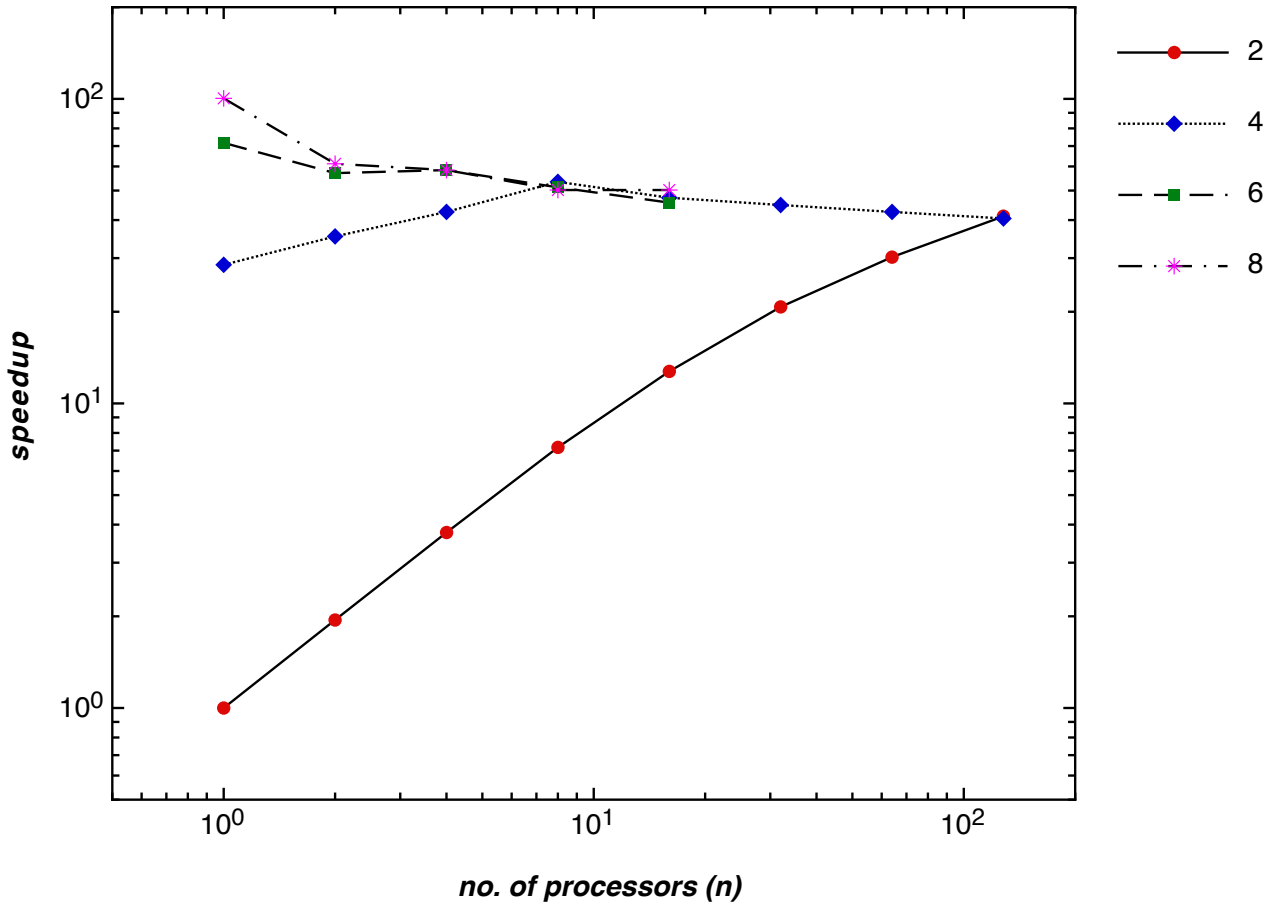

Fig. 3: Speedup as a function of the number of processors for a fixed level of error, for various finite-difference orders. Speedup is defined relative to the second-order code running on a single processor. It is clear that the second-order method scales very well comparatively.

It is also interesting to note the value of n i.e. the number of processors when the different methods begin to compare well in performance. A quick look at Fig. 3 suggests that for the second-order case this occurs in the ballpark of $n = 100$. This value agrees quite well with our estimate from Section 2, wherein we argued that this should happen at $n \sim \frac{N^{1-2/p}}{p} \approx 50^2$. Thus, at least for the case under consideration, it *only takes a hundred processors for the second-order method to achieve comparable performance to the higher-order methods*. This is fairly modest from the perspective of most modern clusters, even those of relatively small size.

In Fig. 4 we show the speedup as a function of the error level for multiple order methods. The *speedup is defined relative to the second-order code running on a single processor, and it is obtained by choosing the value of n for the least wall clock time*. At low accuracy (right side of the graph) one can see that the higher-order methods (4, 6, 8) deliver a significant benefit over the second-order method. However, for high accuracy (left side of the plot) they all deliver *comparable* performance. As argued in the previous section, this is due to the fact that the second-order method exhibits much better parallel scaling behavior throughout (because of the much larger grid sizes required for the same level of discretization error). Note that the second-order method graph starts with a single processor on the extreme-right to nearly 500 on the extreme-left. The eighth-order method is on a single processor throughout (because it actually performs worse with multiple processors, as expected). The sixth and fourth order cases are on a single processor as one goes from right to left decreasing the error level, however, at some point they begin to scale better (due to the rapid increase in N) and that is why one sees the slope "kink" in their speedup graphs. Note that since that improved scaling appears to match the consistent scaling exhibited by the second-order method throughout, we expect

[2]We choose $p = 4$ because the fourth-order method's speedup compares well to the second-order case here.

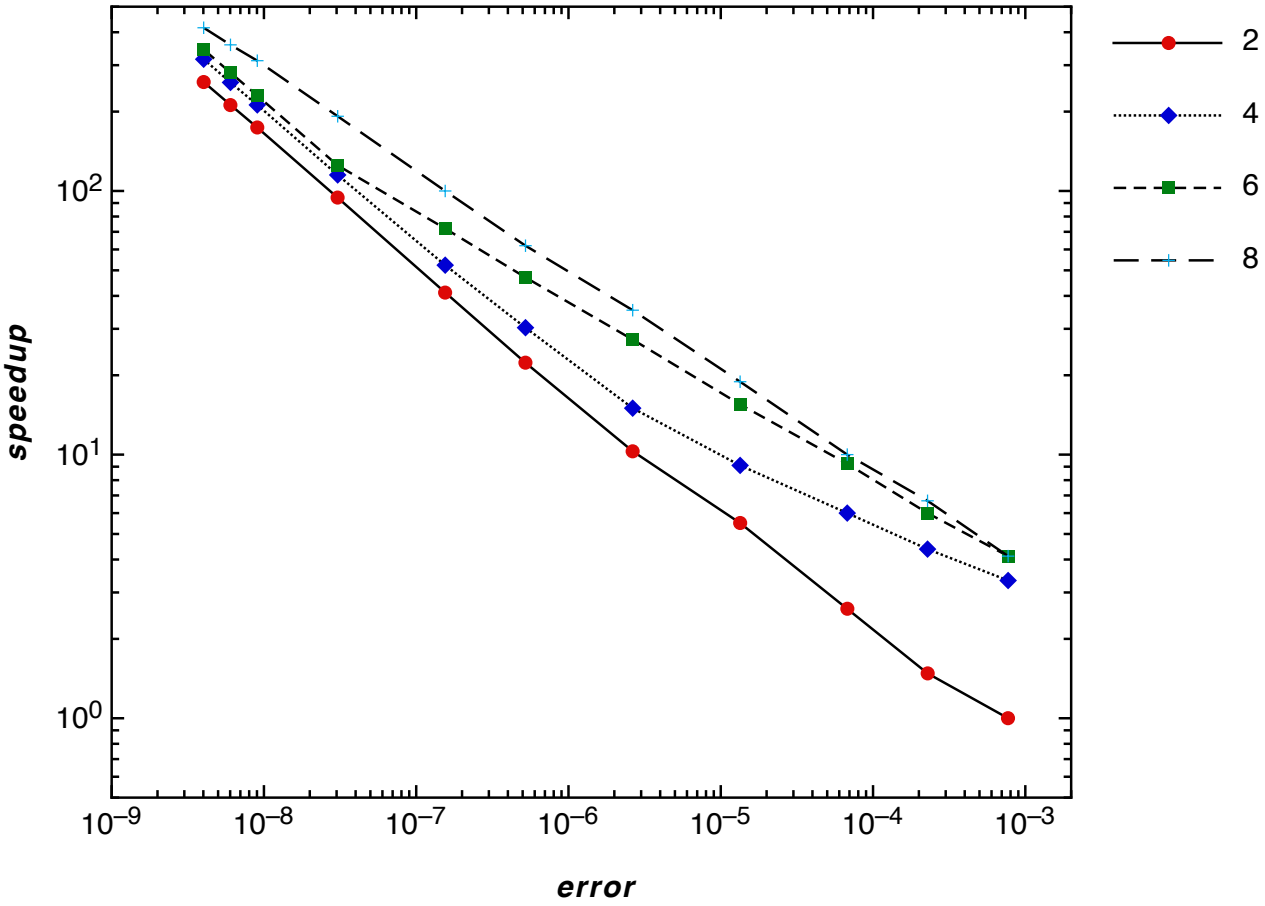

Fig. 4: Best performance (speedup) for various finite-difference orders on a parallel computer cluster. Speedup is defined relative to the second-order code running on a single processor. It is clear that all orders perform comparably for low error levels i.e. on the left side of the graph.

that our main results will hold for error levels even lower than the ones we have tested.

In Fig. 5 we show the same speedup as a function of the method order for multiple error levels. The main point to note again is that *all methods begin to perform comparably for high accuracy computations.*

5. Conclusions

In this work, we have demonstrated that different order finite-difference methods exhibit different scaling behavior on a parallel computer cluster. In general, given a fixed level of accuracy, lower-order methods scale better due to the fact that they require higher resolution and therefore, larger grid sizes. Using a basic example, we have been able to show that *the gain in performance from improved scaling of the second-order method is just enough to have its overall performance match that of a higher-order method.* Of course, the second-order method uses significantly higher computational resources to achieve the same outcome.

Conversely, we have been able to show that a *higher-order method, say, eighth-order, converges so fast that such a method simply does not require any parallel resources at all.* In fact, parallel scaling performance of a higher-order method may be such that one may obtain performance degradation instead of an expected speedup. While we have made our arguments and claims using a simple 1D derivative finite-difference stencil, we expect our main outcomes to hold more generally, including even in 2D and 3D.

To conclude, with the interest of minimizing total execution time and given a sufficiently large computational resource, a "brute force" approach with a lower-order method is likely to perform comparably to a more advanced highly convergent, higher-order method. Since many "real world" science and engineering research codes are written using second-order accurate algorithms, and it is often very challenging to develop algorithms that converge faster, an investment in parallel code development may prove to be quite worthwhile. On the other hand, if one already has a

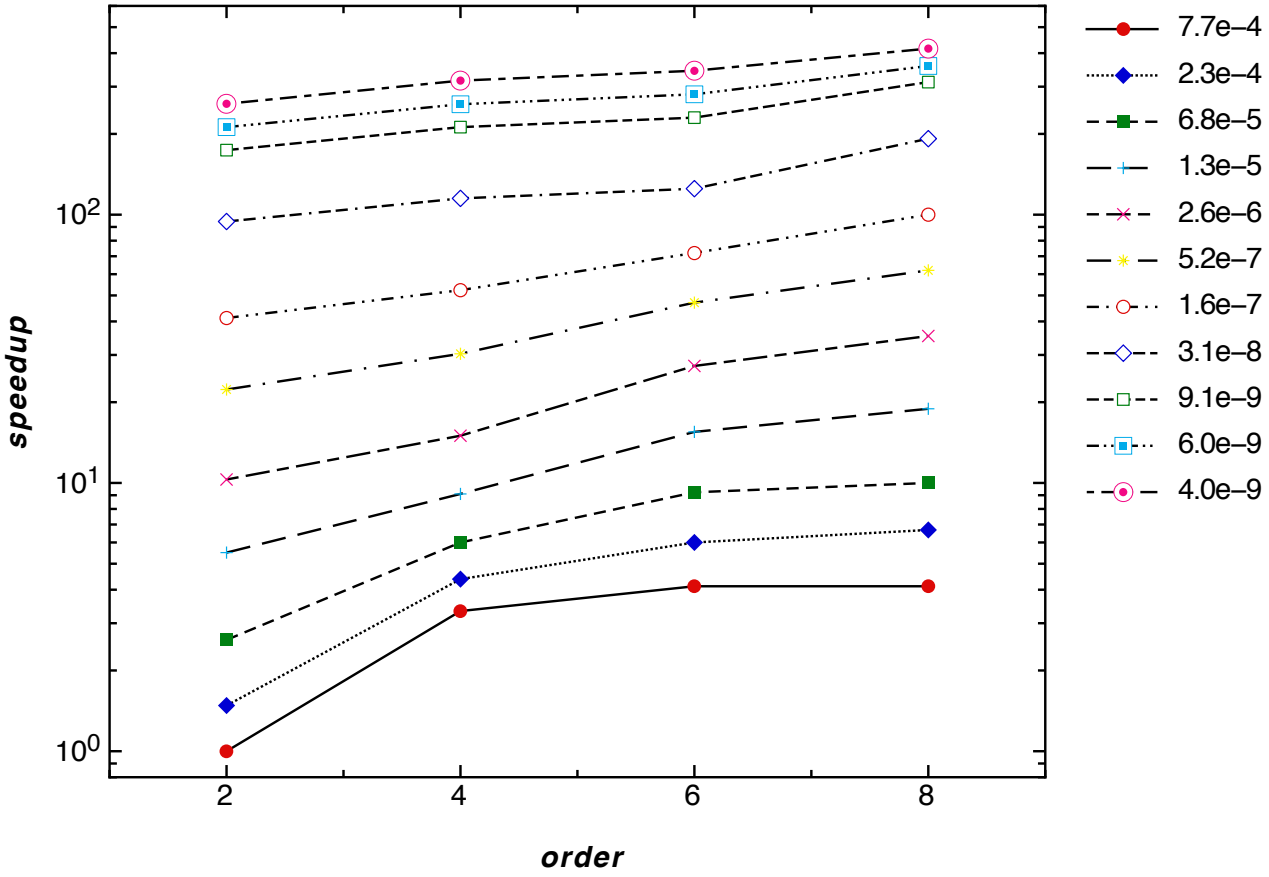

Fig. 5: Same as Fig. 3. Best performance (speedup) for various finite-difference orders on a parallel computer cluster. Speedup is defined relative to the second-order code running on a single processor. It is clear that all orders perform comparably for low error levels.

fast converging serial code then it may be much better to invest in making modest improvements to the convergence rate as opposed to a fully parallel implementation.

6. Acknowledgements

We would like to thank Jay Wang, Alfa Heryudono and Glenn Volkema for feedback on an early draft of this article. G.K. acknowledges research support from NSF Grants No. PHY-1016906, No. CNS-0959382, No. PHY-1135664, and No. PHY-1303724, and from the U.S. Air Force Grant No. FA9550-10-1-0354 and No. 10-RI-CRADA-09.

References

[1] The Top 500 List: http://top500.org/
[2] B. Gustafsson, "High Order Difference Methods for Time Dependent PDE", Springer Series in Computational Mathematics, Volume 38 (2008).
[3] J.S Hesthaven,T. Warburton, " Nodal High-Order Methods on Unstructured Grids: I. Time-Domain Solution of Maxwell's Equations", Journal of Computational Physics, Volume 181, Pages 186-221 (2002).
[4] M. O. Deville, P. F. Fischer, E. H. Mund, "High-Order Methods for Incompressible Fluid Flow", Cambridge University Press (2002).
[5] G. Karniadakis, "High-order splitting methods for the incompressible Navier-Stokes equations", Journal of Computational Physics, Volume 97, Pages 414-443 (1991).
[6] D. Xiu, J. S. Hesthaven, "High-Order Collocation Methods for Differential Equations with Random Inputs", SIAM J. Sci. Comput., 27(3), 1118-1139 (2005).
[7] C-W. Shu, "High-order Finite Difference and Finite Volume WENO Schemes and Discontinuous Galerkin Methods for CFD", International Journal of Computational Fluid Dynamics, Volume 17, Issue 2, Pages 107-118 (2003).
[8] J.T. Beale, " High order accurate vortex methods with explicit velocity kernels", Journal of Computational Physics, Volume 58, Pages 188-208 (1985).
[9] M. R. Visbal, D. V. Gaitonde, "High-Order-Accurate Methods for Complex Unsteady Subsonic Flows", AIAA Journal, Volume 37, No. 10, pp. 1231-1239 (1999).
[10] J. Shen, T. Tang, "Spectral and High-Order Methods with Applications", Mathematics Monograph Series 3 Science Press (2006).
[11] J. Hesthaven, D. Gottlieb, S. Gottlieb, "Spectral Methods for Time-Dependent Problems", Cambridge Monographs on Applied and Computational Mathematics (2007).
[12] OpenMPI website: http://openmpi.org/
[13] http://www.afmc.af.mil/news/story.asp?id=123232827