Contents lists available at ScienceDirect

Journal of Computational Science

journal homepage: www.elsevier.com/locate/jocs

Computation–communication overlap and parameter auto-tuning for scalable parallel 3-D FFT

Sukhyun Song*, Jeffrey K. Hollingsworth

Department of Computer Science, University of Maryland, College Park, United States

A R T I C L E   I N F O

Article history:
Received 16 July 2015
Received in revised form 8 October 2015
Accepted 9 December 2015
Available online 19 December 2015

Keywords:
3-D FFT
MPI
Non-blocking collective
Computation–communication overlap
Auto-tuning

A B S T R A C T

Parallel 3-D FFT is widely used in scientific applications, therefore it is important to achieve high performance on large-scale systems with many thousands of computing cores. This paper describes a new method for scalable high-performance parallel 3-D FFT. We use a 2-D decomposition of 3-D arrays to increase scaling to a large number of cores. In order to achieve high performance, we use non-blocking MPI all-to-all operations and exploit computation-communication overlap. We also auto-tune our 3-D FFT code efficiently in a large parameter space and cope with the complex trade-off in optimizing our code in various system environments. According to experimental results from two systems, our method computes parallel 3-D FFT significantly faster than three existing libraries, and scales well to at least 32,768 compute cores.

1. Introduction

It is important to achieve high performance of the Fast Fourier Transform (FFT) as FFT is widely used in many fields of science and engineering. For example, researchers have recently used three-dimensional FFT (3-D FFT) to run astrophysical *N*-body simulations [1] and turbulent flow simulations [2] on high-performance computing systems. Computing 3-D FFT requires a large number of floating point and memory access operations. So we need a parallel algorithm that can be run on a large-scale system. However, parallel 3-D FFT requires expensive all-to-all communication. Each parallel computing process exchanges a 3-D array with all the other processes. We believe that it would be beneficial to overlap expensive computation and communication in achieving high performance of parallel 3-D FFT.

In this paper, we present a new method for parallel 3-D FFT that can increase scaling to a large number of computing cores in a distributed-memory parallel system, and exploits computation–communication overlap. [1] We design a scalable method by following the 2-D domain decomposition technique [4–8] that decomposes a 3-D array along two dimensions. Accordingly, we can use more computing cores than the approaches presented in [9–11] that use the 1-D domain decomposition and only decompose an array along one dimension. We exploit the overlap of computation and communication by using the non-blocking `MPI_Ialltoall` operation that is described in the MPI-3.0 standard [12]. As the MPI library is widely used in the parallel computing community, it is important to design a parallel 3-D FFT code based on MPI and achieve portability. We use collective communication instead of point-to-point communication due to its better performance and simplicity. The MPI collectives are optimized for complex communication patterns among parallel processes, and we can write a simple code based on collective operations.

Our work is the first to effectively apply non-blocking MPI collectives to the 2-D decomposition technique for parallel 3-D FFT. Popular FFT libraries [4,5,9] do not exploit non-blocking communication. Although the work by Nishtala et al. [8] overlaps computation and communication, it is written in UPC and based on point-to-point communication. Hoefler et al.'s work [10] and our previous work [11] use `MPI_Ialltoall`, but their scalability is limited as they use a 1-D domain decomposition.

Our strategy for computation–communication overlap is to have each process divide an array into multiple small blocks, and repeat computation and non-blocking communication on the divided blocks. In this way, we have opportunities to overlap computation on one block with communication on other blocks, and eventually increase the overall performance. We also increase scaling by applying our overlap strategy to each of the two communication phases in the 2-D decomposition technique for parallel 3-D FFT.

* Corresponding author.
E-mail addresses: shsong@cs.umd.edu (S. Song), hollings@cs.umd.edu (J.K. Hollingsworth).

[1] Note that this paper is an extended version of our previous paper [3], so the two papers share several figures and texts.

http://dx.doi.org/10.1016/j.jocs.2015.12.001
1877-7503/© 2015 Elsevier B.V. All rights reserved.

To optimize the overall performance of our method, we identify and resolve the following requirements. First, *all possible computation steps should be used for the overlap*. Parallel 3-D FFT involves three steps of pre-computation, communication, and post-computation. Simpler approaches given in [8,10,13] can only overlap pre-computation on one data block with communication on other blocks, then perform post-computation on the entire blocks. Instead, we have both pre-computation and post-computation overlapped with communication. Second, *computation should be balanced between the two communication phases*. The 2-D decomposition method involves two phases of all-to-all communication, and there exist three local computation steps before and after each phase. When the communication time is different between two phases, we divide the computation in the middle into two. We then assign a longer computation to the phase with the longer communication. In this way, we increase the opportunity to overlap the computation in the middle with communication. Third, *code should be portable*. One way to ensure fully asynchronous communication for non-blocking MPI collectives is to offload communication processing to special hardware. Instead, we choose another option for asynchronous communication, which calls `MPI_Test` periodically, due to its greater portability. Last, *we should automatically handle the complex trade-off regarding our optimization techniques*. We parameterize our parallel 3-D FFT code and auto-tune the parameters. For example, a data block size for communication and a `MPI_Test` call frequency are set by tunable parameters. Since the parameter space is very large (more than trillions of possible configurations), hand-tuning is not feasible. We utilize the publicly available Active Harmony auto-tuning library [14] and find a good parameter configuration efficiently. We also introduce several techniques to tune our FFT code fast and effectively.

We conducted experiments on two systems at NERSC using up to 32,768 compute cores. Our code performs parallel 3-D FFT faster than three other approaches. Specifically, we achieve a speedup by up to 1.83× over the FFTW library [9], 2.57× over the 2DECOMP&FFT library [5], and 1.95× over the UPC-based code [8].

The rest of the paper is organized as follows. We first describe background information in Section 2. Sections 3 and 4 present how we design and auto-tune our parallel 3-D FFT code. We relate our evaluation and results in Section 5. Section 6 provides the related work. Finally, we conclude and discuss future work in Section 7.

2. Background

This section first reviews FFT computation and describes several basic assumptions underlying our design. We then introduce a general method for parallel 3-D FFT without any computation–communication overlap.

2.1. Fast Fourier Transform

With an input array X of N complex numbers, a one-dimensional FFT produces an output array Y of N complex numbers. When $\omega_N = e^{-(2\pi/N)i}$, $Y[k]$ is defined for $k = 0, 1, \ldots, N-1$ as $Y[k] = \sum_{j=0}^{N-1} X[j]\omega_N{}^{jk}$. The d-dimensional FFT can be computed simply as the composition of a sequence of d sets of 1-D FFTs along each dimension. For example, 3-D FFT for N^3 complex numbers can be computed by three sets of N^2 1-D FFTs along each dimension.

We make the following assumptions in this paper. First, we focus on the *forward* transform that transforms X into Y. Our approach can be easily applied to transform Y backward into X. Second, we only describe the *complex-to-complex* transform that takes an input array of complex numbers and produces an output array of complex numbers. We have also implemented the real-to-complex transform by applying our computation–communication overlap methods, but omit the details of this implementation due to space limitations. Last, we focus on the *in-place* transform and want the output to overwrite the input array. Our approach can be applied directly for the out-of-place transform where the output is written to a separate array.

2.2. 2-D domain decomposition for parallel 3-D FFT

Our design for a parallel 3-D FFT basically follows the *2-D domain decomposition* method. The overall procedure of the 2-D domain decomposition method consists of several steps as shown in Fig. 1. We parallelize 3-D FFT with $\boldsymbol{p}$ processes, and Fig. 1 shows an example of $p = 4$. An input 3-D array is divided equally into $\boldsymbol{p_1}$ sub-arrays along the x dimension and $\boldsymbol{p_2}$ sub-arrays along the y dimension. $p_1 = 2$ and $p_2 = 2$ in Fig. 1, and process numbers are marked on each sub-array. Then each process executes the following steps on each sub-array:

1. **FFTz**: Compute 1-D FFTs along the z dimension. (Assume that elements on the z dimension are adjacent in memory.)
2. **Pack1**: Pack the 3-D sub-array data into a buffer in preparation for all-to-all communication.
3. **A2A1**: Perform blocking all-to-all communication for each group of p_2 processes.
4. **Unpack1**: Unpack the received data into the 3-D sub-array with the new memory layout such that elements on the y dimension are adjacent in memory.
5. **FFTy**: Compute 1-D FFTs along the y dimension.
6. **Pack2**: Pack the 3-D sub-array data into a buffer in preparation for all-to-all communication.
7. **A2A2**: Perform blocking all-to-all communication for each group of p_1 processes.
8. **Unpack2**: Unpack the received data into the 3-D sub-array with the new memory layout such that elements on the x dimension are adjacent in memory.
9. **FFTx**: Compute 1-D FFTs along the x dimension.

We divide the 2-D decomposition procedure into two *communication phases*: **Phase1** includes steps from FFTz through FFTy, and **Phase2** includes the rest of the procedure.

An alternative way to compute a parallel 3-D FFT is to use the *1-D domain decomposition* that only decomposes an array along one dimension rather than two. The 2-D domain decomposition method is more scalable than 1-D decomposition since we can use more computing processes. The 2-D decomposition requires two phases of all-to-all communication, which involves more complex communication pattern than the 1-D decomposition. However, the 2-D decomposition performs communication on a small sub-group of processes while the 1-D decomposition uses a whole set of processes for all-to-all communication. Thus, when both methods can be used, the system environment determines which method is faster. The 1-D decomposition can be considered as two special cases of 2-D decomposition. One case is $p_1 = 1$, where a 3-D array is only decomposed along the y dimension. The other case is $p_1 = p$, where a 3-D array is only decomposed along the x dimension. Note that our approach supports 1-D decomposition through auto-tuning. To achieve high performance parallel 3-D FFT, we tune the value of p_1 and automatically choose between a 1-D decomposition and a 2-D decomposition.

3. Parallel 3-D FFT

This section describes the design of our new parallel 3-D FFT algorithm. We will show how we can improve the 2-D