ABSTRACT

We present a method to design and auto-tune a new parallel 3-D FFT code using the non-blocking MPI all-to-all operation. Preliminary results show that we maximize computation-communication overlap, and execute 3-D FFT faster than the MPI-enabled FFTW library by up to 1.76x.

1. INTRODUCTION

The Fast Fourier Transform (FFT) is widely used in many fields of science and engineering. Specifically, scientific applications have recently used 3-D FFT to run astrophysical N-body simulations [3] and blood flow simulations. Parallel 3-D FFT in distributed-memory parallel systems require all-to-all communication. Researchers [4, 1] tried to increase the 3-D FFT performance by overlapping computation and communication with the non-blocking MPI_Ialltoall operation described in the new MPI-3.0 standard. However, the prior overlap approaches have several limitations. Kandalla et. al [4] implement the overlap between multiple independent input arrays. This approach is not effective in many applications as scientific simulations [3] normally perform successive 3-D FFT operations on a single array. Although Hoefler et. al [1] overlap computation and communication on a single array, their implementation is not optimized.

We design a new 3-D FFT code with MPI_Ialltoall to achieve higher performance than the prior approaches. Our approach is to divide an input array into multiple small blocks and overlap computation on one block with communication on other blocks. There are several unique characteristics of this work. First, we optimize computation-communication overlap. We hide communication behind as much computation as possible. Second, we design a portable code. We use MPI_Test for fully asynchronous communication [2] rather than rely on special hardware support or separate threads. Third, we also optimize local computation. We improve cache performance through loop tiling. Last, we parameterize our code and auto-tune the parameters. So we can cope with the complex trade-off regarding our optimization techniques.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC ’13
Copyright 2013 ACM X-XXXX-XX-X/XX/XX ...$15.00.

2. NEW DESIGN OF PARALLEL 3-D FFT

Figure 1 shows the overall procedure of our parallel 3-D FFT algorithm. We parallelize 3-D FFT with p processes rank0, rank1, ..., rank(p−1) (p = 2 in Figure 1). An input 3-D array is divided equally into p blocks along the x dimension and assigned to each process. Note that 3-D FFT can be computed simply as the composition of a sequence of three sets of 1-D FFTs along each dimension. We rely on the FFTW library for optimized local 1-D FFT computation. Each process first executes the following two steps:
1) FFTz: Compute 1-D FFTs along the z dimension.
2) Transpose: Change the memory layout for the next step, so that elements on the y dimension are adjacent in memory.

Then an input array is divided into multiple small blocks called communication tiles along the z dimension as shown with a colored box in Figure 1. Each process executes the followings steps repeatedly for each communication tile:
3) FFTy: Compute 1-D FFTs along the y dimension.
4) Pack: Pack the 3-D array data into a buffer in preparation for all-to-all communication.
5) A2A: Perform non-blocking all-to-all communication.
6) Unpack: Unpack the received data into the 3-D array so that elements on the x dimension can be adjacent in memory.
7) FFTx: Compute 1-D FFTs along the x dimension.

In this way, we can overlap computations (FFTy, Pack, Unpack, and FFTx) on one tile with communication (A2A) for other tiles. For example, while FFTy and Pack are executed on a tile, A2A for the previous tile can be performed. Also, when Unpack and FFTx are executed on a tile, the communication for the next tile can be performed.

To make an auto-tunable code, we define ten parameters in our algorithm. We first have a communication tile size parameter (T in Figure 1). A window size parameter is used to define the degree of communication parallelism. To ensure progress of the all-to-all communication without any hardware support, we have four frequency parameters of MPI_Test calls. To optimize computations with respect to cache reuse, we use loop tiling inside each communication tile, for which we have four more parameters.
3. AUTO-TUNING METHOD

Figure 2: Auto-Tuning Procedure

Active Harmony (AH) [6] is a general software framework to auto-tune user-specified parameters for a tunable code. Figure 2 shows the overall procedure of how AH interacts with our code. The AH server provides the AH client with a parameter configuration to be tested. Then the client executes 3-D FFT with the received configuration and reports the performance back to the server. This procedure is repeated until the server finds the best configuration. The AH server uses the Nelder-Mead (NM) optimization method [5] to search for the best parameter configuration.

There are several techniques we introduce for effective and fast auto-tuning. First, we penalize an infeasible configuration. Since the parameter space can be irregular and shaped in a non-rectangle, the AH server can provide an infeasible parameter configuration. In this case, the AH client reports the worst performance value (infinity) immediately back to the server, so that we can avoid executing an infeasible configuration. The second technique is to reuse the prior performance data. Since we use NM in the discrete integer domain of parameters, the AH server can provide the same configuration even though it has been already tested before. To save tuning time, the AH client maintains and utilizes the history of tested configurations. Third, to improve the auto-tuning speed, we skip executing FFTz and Transpose as those steps are independent of the ten parameters. Last, instead of searching a whole set of all possible values of a parameter, we reduce a search space to a log scale and consider power-of-two values for testing.

4. EVALUATION

We compare three different methods for parallel 3-D FFT. 

FFTW is the MPI-enabled FFTW library. NEW is our method described in Sections 2 and 3. TH is Hoefler et. al’s method [1] that also implements computation-communication overlap. For fair comparison, we slightly modified the original code of TH and auto-tuned it similarly to NEW.

Figure 3 shows the speedup of NEW and TH over FFTW on the Hopper system, a Cray XE6 machine at NERSC. $p$ means the number of parallel computing processes. $N$ is the number of elements on each dimension in a 3-D input array of $N^3$ complex numbers. The speedup of NEW over FFTW ranges from 1.48× to 1.76×. On the other hand, TH shows the maximum speedup of 1.12×, and there is even a setting such that TH is worse than FFTW.

Figure 4: Performance Breakdown

In Figure 4, we break down the performance of NEW and TH for the $p = 256$ and $N = 2048$ configuration. NEW-0 is the non-overlapped version of NEW. Likewise, TH-0 is the non-overlapped version of TH. While NEW-0 takes a long time for A2A (marked with Wait), NEW reduces the A2A time down to almost zero, which means NEW nearly achieves the perfect computation-communication overlap. This high degree of overlap explains why NEW is the fastest. TH performs a low degree of overlap as shown in Figure 4. Since FFTW does not exploit non-blocking communication, the performance should be similar to NEW-0.

5. REFERENCES


