Redistribution strategies for portable parallel FFT: A case study

Anshu Dubey
Astronomy & Astrophysics
University of Chicago
5640 S. Ellis Ave.
Chicago, IL 60637
E-mail: dubey@tagore.uchicago.edu
http://astro.uchicago.edu/rranch/Computing

Daniele Tessera
Dipartimento di Informatica e Sistemistica
Università degli Studi di Pavia
I-27100 Pavia
Italy
E-mail: tessera@gilda.unipv.it
http://mafalda.unipv.it

Abstract

The best approach to parallelize multidimensional FFT algorithms has long been under debate. Distributed transposes are widely used, but they also vary in communication policies and hence performance. In this work we analyze the impact of different redistribution strategies on the performance of parallel FFT, on various machine architectures. We found that some redistribution strategies were consistently superior, while some others were unexpectedly inferior. An in-depth investigation into the reasons for this behavior is included in this work.

Introduction

The performance of a parallel FFT algorithm depends on its scalar performance, efficiency of communication and load balance. Of these the best approach to communication efficiency is most debated, since near optimal load balance [15] and scalar performance are easily achieved (a comprehensive list of fast scalar FFT's can be found at http://www.fftw.org). In this paper we consider the impact of different communication approaches on the performance of the parallel FFT's in an attempt to identify strategies that perform uniformly well on different platforms.

An FFT algorithm presents a wide array of choices for parallelization. One can use transpose based FFT's as opposed to the distributed ones. In a transpose based FFT, at least one data dimension resides within the nodes at all times. The data distribution typically starts with at least one dimension local and one or more of the other dimensions parallel. The local dimension is transformed within the node, followed by a transpose. The transpose could be local if the next dimension is sequential, or distributed if it is parallel. The transform itself is again local. The same process is repeated for the third dimension if it exists. The distributed FFTs place no locality restriction on any dimension. Here each dimension can be parallel, and pair-wise data exchange takes place between nodes whenever non-local data are needed for the transform [6]. As a result, communication is interspersed with computation for each parallel dimension, and the optimized scalar FFT routines cannot be used. The range of communication is shorter in this method, but the volume of data transferred is higher. Within each of these two broad categories there are several alternatives for data distribution, and interprocessor communications [3, 11]. We concentrate on the second category, since we consistently found distributed FFT's to be slower than the transpose based ones, as also indicated in [7]. The distributed FFT's are useful only when the data distribution constraints are favorable to them, for example when every dimension of the transform is parallel, or when the data are distributed in block scattered fashion [6]. The typical data distribution for applications using the parallel FFT is one where at least one of two or three dimensions is local. Without any loss of generality, here we consider a three dimensional distribution with the third dimension distributed over the processors.

The communication costs of the transpose based FFT's are limited to the distributed transpose. The transpose itself can be based on different communication policies such as collective versus point-to-point communications, blocking versus non blocking protocols. We have incorporated these variations in the different kernels of the distributed transpose to study their impact on the performance with different machine architectures. All the implementations use MPI based communications, in the interest of portability [8, 12]. They were analyzed using Medea [5], a tool for workload characterization and performance diagnosis of parallel applications.

This work has a wider implication. Its results are applicable to codes that have large volumes of structured data transfer [4]. The results also pin point the causes for the loss in performance, and show some surprising deviations from theoretical expectations.

Analysis Strategy

The parallel performance of a code is dependent upon some factors which are inherent in a machine (topology of the interconnecting networks, latency, bandwidth, routing strategies and efficiency of the underlying communication protocols), and some which are under the control of the application programmers. These are: selection of appropriate protocols (e.g. point-to-point vs collective, blocking vs non-blocking communications in MPI), and scheduling the data transfers to maximally utilize the available bandwidth. For this study we have created five different kernels of the distributed transpose (and hence parallel FFT) that highlight these variations in protocol and transfer schedules.

The kernels were run on two different machines and their behavior was analyzed. The machines used in this study are IBM Sp2 and Cray T3E. They represent two very different architectures commonly used in the parallel computing framework. They differ in all of the basic parallel paradigms; memory addressing, interconnection networks, communication models, routing strategies and the system software controlling individual nodes. The IBM Sp2 is a distributed memory machine with message passing as the communication model. It has multi stage interconnecting network with a fixed multi path routing [1, 14]. Each node runs a full operating system (AIX 4.1/POE) as its system software. On the other hand, the Cray T3E has virtual shared memory as its communication model. The interconnection network is a three dimensional torus with multiple path adaptive routing [13]. A microkernel (UNICOS/mk) controls the running of individual nodes. In addition to these hardware and software differences, the load of the two systems, i.e, simultaneously active parallel runs and interactive users, is very different and leads to different communication characteristics.There are several studies addressing specific machine characteristics [10, 16, 2]. In this work we have ignored the machine peculiarities, instead we have focused on the effective exploitation of potentialities of both the machines and the communication strategies.

All the FFT kernels use MPI-1 for portability. There is no performance loss from this choice since both machines have extremely efficient implementations of MPI. All the kernels have the same initial and final data distributions though the two distributions differ (we have not included the additional transpose needed to restore the data distribution). The amount of computation and the total volume of data transfer are also identical for all the kernels. The difference is in the number of MPI calls and scheduling of the data transfers. Optimized local FFT's were used in both the machines with complete load balance at all times, to ensure best possible performance for each individual implementation. Each of them has been instrumented by adding probes to monitor various activities (i.e., communications, computations, and data redistributions). These probes acquire detailed measurements about the monitored activities and collect them into tracefiles. Special care has been taken to minimize the perturbation in the kernel performance due to the monitoring activity. Tracefiles are then processed by Medea to derive high level performance metrics, like speedup and computation/communication times. The timing reflects 10 iterations per run; each iteration consisting of one forward and one inverse FFT. The data distribution at the end of one iteration is identical to the data distribution at the beginning of the iteration (the inverse FFT performs another transpose that restores the data distribution). We have taken special care to mimic the performance of our FFT kernels to large production runs where the start up costs become negligible since they are amortized over a very large number of iterations.

FFT Kernels

In this section we give the details of the five different kernels used in this study. We define a three dimensional problem of size of N^3, distributed over P processors. The domain decomposition is slab wise; where only one dimension is parallel at a time, and each processor has N/P planes. The planes are transformed locally, followed by a distributed transpose, followed by a transform in the third dimension. The initial data distribution, the final data distribution and the volume and range of data transfer are identical for all the five kernels. The distributed transpose is essentially a complete exchange[4]. It can be done with a single complete exchange step, where all planes are processed together for communication, or in N/P steps, where each plane initiates its own complete exchange.

The second alternative can theoretically permit the overlap of communication with computation. The complete exchange itself can be carried out with either point-to-point communications, or with collective communications such as all-to-all. The point-to-point communications permit more flexibility in the code, since they can be scheduled by the user, and can be used in a non-blocking mode.

Each kernel has been named based upon its communication mode. The first kernel is called *The Replace Kernel*, since it uses **send_receive_replace** mode of communication. This is a point-to-point blocking mode that uses the same buffer for both send and receive. The contents of the buffer are replaced by the received data after the send is completed. This kernel processes one plane at a time, resulting in $(P-1) \times N/P$ calls per node ($(P-1)$ calls per plane). The second kernel, called *The Standard Kernel* uses the standard **send** and **receive** calls for communication. This is also a point-to-point blocking mode of communication. The difference from the first kernel is that the send and receives can be scheduled separately and can use different buffers. The number of communication calls issued is twice that of the first kernel. The third kernel, *The Overlap Kernel*, uses non-blocking send and receive calls in it. Thus by scheduling them carefully it is theoretically possible to overlap communication with computation, hence better performance. In every other respect, this kernel is identical to the second one. The first three kernels use point-to-point communications. The fourth kernel is *The Oneplane Kernel* which uses collective communication **all_to_all** for each plane separately. The number of communication calls issued per node is N/P. The send and receive buffers are different. This kernel could have exploited the communication and computation overlap, if a non-blocking all_to_all call had been available. The fifth and final kernel, the *Allplanes Kernel*, uses the collective communication for all the data on the node at once. Since only one communication call is issued per node, this kernel has minimum setup overhead. We have not included the kernels that process all planes together in point-to-point communications, since they sometimes result in deadlocks.

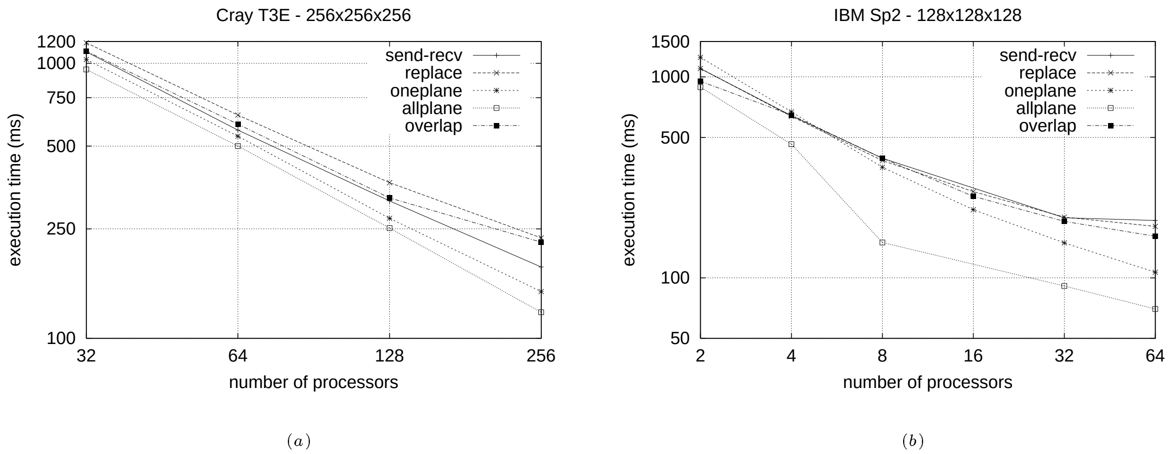
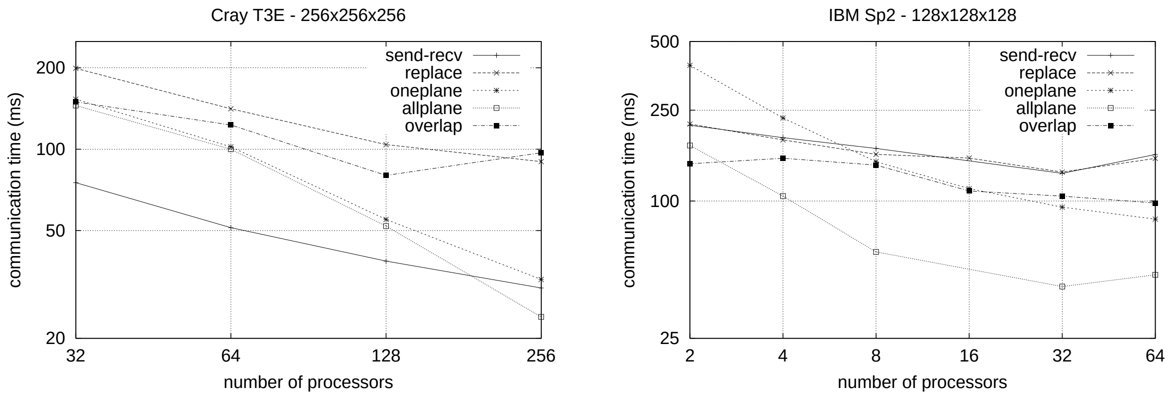


Figure 1: Execution times, in log-log scale, for a single FFT step, as a function of the number of allocated processors, on a Cray T3E (a) and an IBM Sp2 (b).

Performance Results

In this section we analyze the performance of the FFT kernels described in the previous section on both Cray T3E and IBM Sp2. As a preliminary overview of the kernel performance, Figure 1 plots the wallclock execution time of a single forward and inverse FFT step as a function of the number of allocated processors on both the machines. Note that, to improve their readability, these figures have been plotted using a log-log scale.

The wallclock time includes all communication and computation timings except the initial startup times (e.g., broadcast of the problem grid size and its decomposition, initialization of the scalar one/two dimensional FFT solvers). Figure 2 shows the time spent by each processor, in communication activities for each of executions depicted in Fig. 1. To improve their readability, also these figures use a log-log scale. Since we are interested in comparison of different kernels rather than the parallel machines, the data sizes and number of processors differ on the two machines. This is mainly because the machines themselves differ in the number of available processors and the amount of memory available per node. Hence the executions on the Cray T3E solve a grid of $256\times256\times256$ points with a number of processors ranging from 32 up to 256 while the IBM Sp2 executions solve a grid of $128\times128\times128$ points with up to 64 processors.



(a) (b)

Figure 2: Communication times, in log-log scale, for a single FFT step, as a function of the number of allocated processors, on a Cray T3E (a) and an IBM Sp2 (b).

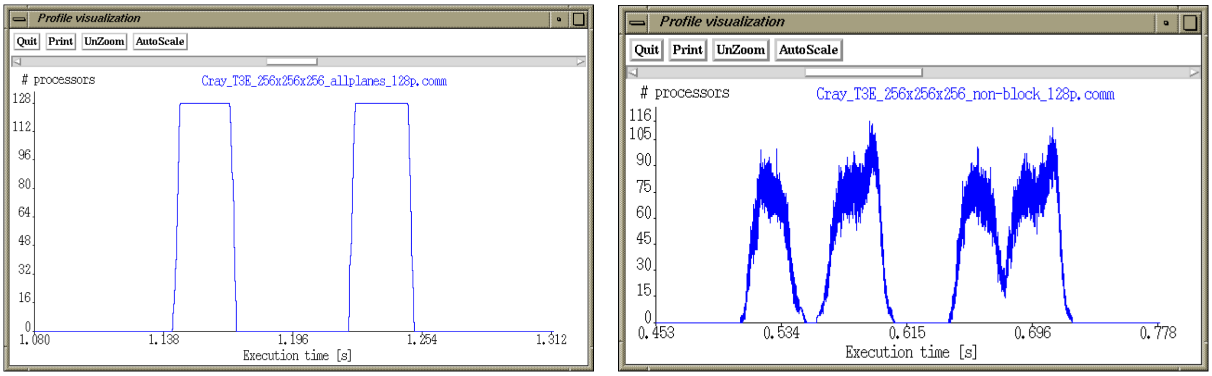
A few observations can be drawn from these figures:

1. All FFT kernels scale quite well on both the machines even with relatively large number of processors.
2. The performances of the kernels relative to each other are similar in both the machines, despite their architectural differences.
3. Minimizing the communication time does not lead to minimum execution time.
4. Overlapping computation and communication may results in reducing performance rather than enhancing it.

While there is a strong evidence of performance portability from one machine to another, the relative performance of the FFT kernels is somewhat unexpected. The rule of thumb that minimizing the communication time will result in maximizing the performance of parallel codes does not always apply. For example, if we compare the total execution time on the Cray T3E (Fig. 1(a)) with its communication counterpart (Fig. 2(a)), we can see that the Standard Kernel (with point-to-point communication) has the minimum communication time but its total execution time is somewhere in-between. This curious effect is because the time spent in MPI calls is just a part of the cost of managing the communication activities.

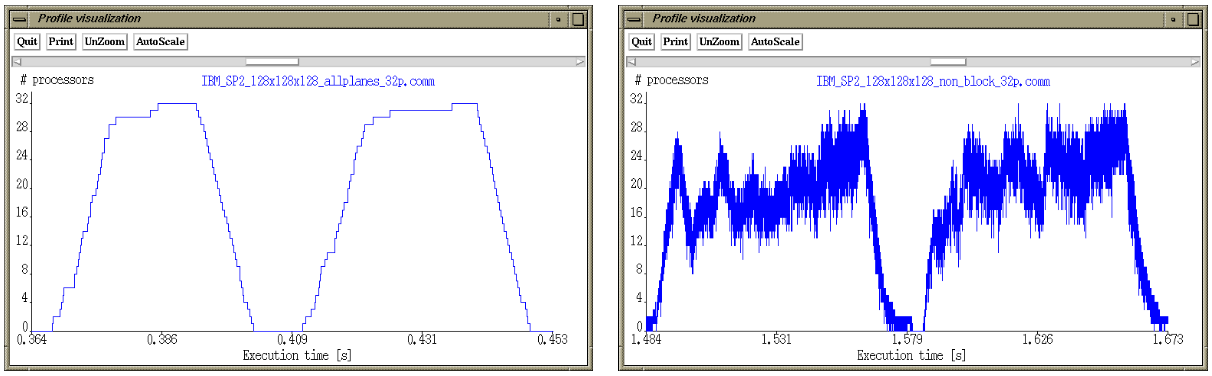
Another surprising observation is that one of the worst performing kernel is the one with overlapping communication and computation, while the best performing one is the one with a single `MPI_Alltoall` call. Given the amount of flexibility available in scheduling the data transfers, the outcome was expected to be different. The Cray T3E does not have a separate processor for communication, hence the lack of performance gain in the overlap kernel is not surprising. However, the actual loss in performance is unexpected. It is even more surprising to see performance loss, and its extent in the Overlap kernel on the IBM Sp2 machine, since it does have separate communication processors. For instance, the FFT of a 128×128×128 running on 32 IBM Sp2 processors takes $191ms$ for the overlap kernel, while the Allplanes kernel takes $91ms$ only. Also, the difference in timing is not entirely due to communication activities, which is $105ms$ for the Overlap kernel and $42ms$ for the Allplanes. That still leaves $37ms$ timing difference unaccounted for. There is very curious explanation for this which is explained later in the section.

To determine the software costs involved is message passing, we analyzed the performance data using Medea. Note that by communication time we do not mean time to physically deliver the message to the destination, but the time spent by the processors to handle the communications (that is, performing MPI functions). Several tasks have to be accomplished by both the sender/receiver processors. The sender has to copy the message from the program memory space to system buffers. The issuing process is blocked until the message is either buffered or sent. This can result in delay if the system buffers are full. For small messages usually eager protocols, which do not require special care from the sender, are used. Unfortunately only a very limited number of small messages can be dispatched with eager protocols due to buffering limits. The requirements of real life applications do not allow communication policies based on eager protocols only but require rendezvous protocols which require a special handling of outcoming messages. The sender has to ensure that there is enough buffer space in the receiver processor before sending the data. Hence it first sends a system message (usually named "envelope") to the destination for reserving the buffer. The receiver can make the reservation or defer the sending until it has enough space. When the sender receives the acknowledgement, it sends the message itself (usually referred as the payload). The receiver has



(a) (b)

Figure 3: Communication profiles for allplane (a) and overlap (b) kernels solving a grid of 256×256×256 points with 128 processors of a Cray T3E.



(a) (b)

Figure 4: Communication profiles for IBM Sp2 for allplane (a) and overlap (b) kernels solving a grid of 128×128×128 points with 32 IBM Sp2 processors.

to move data from system to application buffer after the message arrives. It is obvious that when a lot of messages are sent to a processor it quickly consumes all the buffering space and starts to defer communications.

This approach to message passing is very general and is deep inside the formal properties of message passing paradigms. It is true also for Cray T3E, even though there is a virtual shared memory on top of that. The cost of managing the communication is therefore directly linked to the number of MPI calls. Even in the ideal case with no system buffer contentions, the communication software has to schedule few system tasks which manage the communication. These are executed concurrently with the application and their impact on the overall performance may not be negligible.

The situation is worse with non-blocking point-to-point communications as illustrated by Figures 3 and 4. These figures show the impact of communication by plotting the number of processors involved in communication activities as a function of the execution time on both the Cray T3E and IBM Sp2 respectively. Figures 3 (a) and 4 (a) show the time behavior of the processors performing the collective `MPI_Alltoall` data redistribution while Fig. 3 (b) and 4 (b) show the processors involved in point-to-point non blocking communications (i.e., `MPI_ISend`, `MPI_Irecv`, and `MPI_Wait`). The communication profiles show that processors are somewhat synchronized in the collective communication, whereas in the overlap kernel there are a lot of fluctuations. This is because the non-blocking calls are costlier in their setup time, and need additional buffering and talking to the hardware. Even when sophisticated hardware allows physical overlap of computation and communication activities, there is very little real benefit. For example, on the IBM Sp2 the communication processor is able to address the operating system buffers and to move data from/to them directly by using DMA techniques. Hence, theoretically the main processor may execute the application code concurrently with the communication activities. In truth, memory contentions limit the real benefits in terms of overall execution time. Up to 70% of the memory bus bandwidth may be used up by the communication processor, leaving very limited scope for concurrent processing by the main processor. Some curious results come up because of this complexity. For instance, if we compare the overall performance in Figure 1 with the corresponding communication times in Figure 2, we find that the difference in the overall performance of the overlap and standard kernels cannot be explained by the communication time alone, since in some cases the communication time is actually lower. This is also true of the missing $37ms$ earlier in the section. This is due to the hidden cost of non blocking communication, where the control is returned to the application before the actual physical data transfer is complete. This data transfer steals memory bandwidth from the main processor, thus the time of actual data transfer gets added to the computation rather than communication. Thus in the example of the overlap and Allplanes executions with 32 processors of an IBM Sp2, the computation time increases from the $49ms$ of Allplanes kernel to $86ms$ of the overlap one, accounting for the missing $37ms$.

Performance Modeling

By modeling the communication and computation performance of the FFT kernels, we could gain enough insight to predict their behavior. For this we investigate the relationship of computation and communication times of a single FFT step with the number of allocated processors and the data redistribution policy. Further, we parameterize the analytical models by applying numerical fitting to the timings. Figure 5(a) and (b) show the measured times (i.e., the points) and the corresponding fitting model (i.e., the contiguous line) for communication and computation phases respectively. The times depicted in the figure are expressed in ms and are related to a single iteration of the Allplanes FFT kernel reported as a function of the number of allocated processors on the Cray T3E with a problem size of 256×256×256 points. Numerical fitting captures the scalability of the Allplanes kernel. The analytical expression for the communication times is:

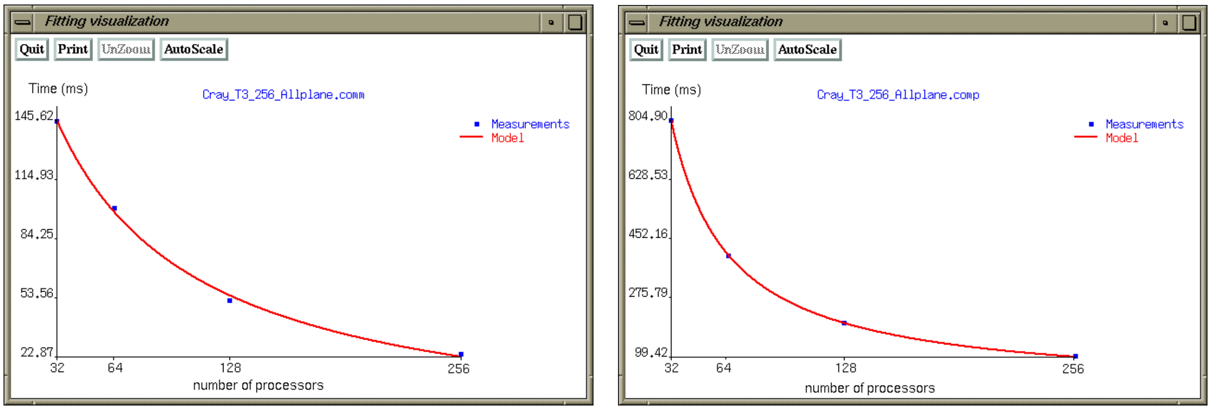
$$t_{COMM} = a_0 + \frac{a_1}{a_2 + np}$$

where:

$$\begin{aligned} a_0 &= \text{-}26.527 \\ a_1 &= 13210.143 \\ a_2 &= 47.447 \\ 32 \leq\ & np \leq 256 \end{aligned}$$

Parameters a_0 and a_2 together represent the latency, while a_1 represents the amount of exchanged data, and np is the number of processors. Here we are looking at the average time spent by the allocated processors in communication activities and hence the latency also include the software costs, which usually overwhelm their corresponding hardware costs. A close to inversely proportional relation between communication time and the number of allocated processors reflects the good scalability of the Allplanes kernel.

The time spent by the processors in computation can be expressed by:



(a) (b)

Figure 5: Communication and computation times together with their corresponding fitting curves for a FFT step on the Cray T3E.

$$t_{COMP} = a_0 + \frac{a_1}{np}$$

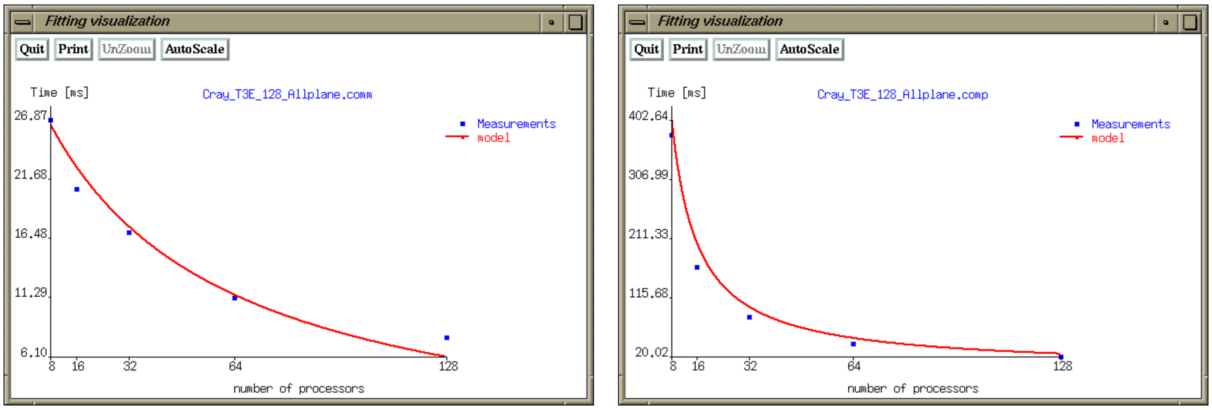
where:

$$a_0 = \text{-}1.278$$
$$a_1 = 25779.423$$
$$32 \leq np \leq 256$$

Here a_1 represents the amount of computation required by the kernel, np is the number of processors. a_0 is a tuning parameter used to improve the numerical accuracy of the fitting function and to express the amount of sequential computation in the kernel; it plays a very limited role since it is more than four magnitude order less than a_1. There is a stronger inverse relationship with the number of processors in the computation time, since there is very little inherently scalar content in the FFT kernels.

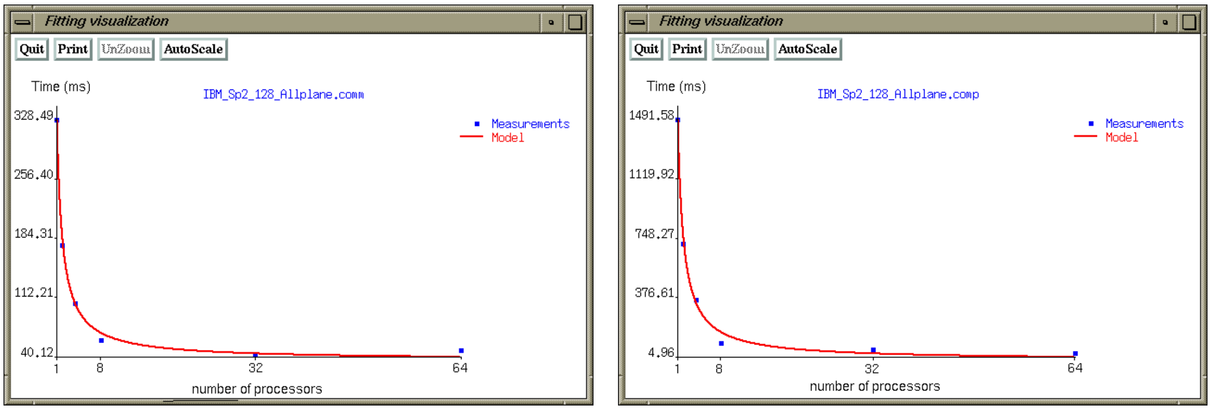
The accuracy of numerical modeling can be verified by applying them to different problem sizes. Figure 6 shows the estimations (contiguous line) and the measured times (points) for the Allplane kernel executions on the Cray T3E with a 128×128×128 problem. Figure 6 (a) is related to the communication time while Fig. 6 (b) to its computation counterpart.

Note that the model predictions are very accurate for all the measured execution but the communication time for the execution with 128 processors. In this case, the error (about 25% of overestimation for the model) on the computation time is mainly due to the better cache exploitation on the data transfers on a smaller grid. This is a case limit, because a problem of 128×128×128 points can be subdivided in at maximum 128 portions (i.e., each processor has a single plane).



(a) (b)

Figure 6: Communication and computation predictions together with their corresponding measures for the Alltoall kernel on the Cray T3E.



(a) (b)

Figure 7: Communication and computation times together with their corresponding fitting curves for a FFT step on the IBM Sp2.

Similar results have been derived from the executions on the IBM Sp2. Fig. 7 (a) and (b) show the fitting curves for respectively the communication and computation times.

The analytical expression for the communication curve is the same as that for the Cray T3E, with difference in the value of parameters. The fitting function for the computation times is somewhat different on the two machines. The expression for IBM Sp2 is :

$$t_{COMP} = a_0 + \frac{a_1}{a_2 + np}$$

where:

$$a_0 = \text{-}16.66$$
$$a_1 = 1381.833$$
$$a_2 = \text{-}0.084$$
$$1 \leq np \leq 64$$

The difference is in the a_2 parameter which is mainly a tuning parameter used by the fitting algorithm to improve the numerical accuracy of the fitting function.

The basis structure of the models is very similar on both the machines, despite their architectural differences. These models reiterate our observations that major characteristics of the kernels depend more on the communication policies than the machine architectures.

Conclusions

The work reported in this paper clearly indicates that there are some type of communication protocols which are superior to others for a particular application type. This holds true for different parallel machines with hardly any common architecture characteristics. The communication libraries are designed for the so called loosely coupled applications. Hence benchmarks like ping-pong timings [9] do not present the real picture of the performance as seen in real life applications. These applications are much more complex, and have unexpected consequences from resource contentions, since most often they stretch all available resources to limit. Hence it is quite likely that superiority of a communication policy may be more application specific than machine or protocol specific. Also, some theoretically superior protocols may actually do worse due to contentions and overheads.

Acknowledgements

This research, conducted in part at the Maui High Performance Computing Center, and NASA Goddard Space Flight Center, was sponsored in part by the Air Force Research Laboratory, Air Force Materiel Command, USAF, under cooperative agreement number UNIVY-0282-U00, and NASA's HPCC initiative. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Research Laboratory, the U.S. Government, The University of New Mexico, or the Maui High Performance Computing Center.

References

[1] T. Agerwale, J. Martin, J. Mirza, D̃. Sadler, D. Dias, and M. Snir. SP2 system architecture. *IBM System Journal*, 34(2):152–184, 1995.

[2] Ed C. Anderson, J. Brooks, C. Gassi, and S. Scott. Performance analysis of the T3E multiprocessor. In *Proceedings of Supercomputing'97.* ACM Press and IEEE Computer Society Press, 1997.

[3] D.H. Bailey and P.O. Frederickson. Performance results for two of the NAS parallel benchmarks. In IEEE, editor, *Proceedings of Supercomputing'91*, pages 166–173. IEEE Computer Society Press, 1991.

[4] S. Bokhari. Multiphase Complete Exchange: A Theoretical Analisys. *IEEE Transactions on Computer*, 45(2):220–229, 1996.

[5] M. Calzarossa, L. Massari, A. Merlo, M. Pantano, and D. Tessera. MEDEA – A Tool for Workload Characterization of Parallel Systems. *IEEE Parallel and Distributed Technology*, 2(4):72–80, 1995.

[6] A. Dubey, M. Zubair, and C.E. Grosch. A general purpose subroutine for fast Fourier transform on a distributed memory parallel machine. *Parallel Computing*, 20(12):1697–1710, 1994.

[7] I. Foster and P.H. Worley. Parallel algorithms for the spectral transform method. *SIAM Journal on Scientific Computing*, 18(3):806–837, 1997.

[8] William Gropp, Ewing Lusk, Nathan Doss, and Anthony Skjellum. High-performance, portable implementation of the MPI Message Passing Interface Standard. *Parallel Computing*, 22(6):789–828, 1996.

[9] R. Hockney and M. Berry. Public International Benchmarks for Parallel Computers: PARKBENCH Committee Report-1. *Scientific Computing*, 3(2):101–146, 1994.

[10] K. Hwang, Z. Xu, and M. Arakawa. Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing. *IEEE Transactions on Parallel and Distributed Systems*, 7(5):522–535, 1996.

[11] S.L. Johnsson and R.L. Krawitz. Communication Efficient Multiprocessor FFT. Technical Report TR-25-91, Division of Applied Sciences - Harvard University, 1991.

[12] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3/4):165–414, 1994.

[13] S. Scott. Synchronization and communication in the T3E multiprocessor. *ACM SIGPLAN Notices*, 31(9):26–36, 1996.

[14] M. Snir, Hochschildm P., D. Frye, and K. Gildea. The communication software and parallel environment of the IBM SP2. *IBM System Journal*, 34(2):205–221, 1995.

[15] P.N. Swarztrauber. Multiprocessor FFTs. *Parallel Computing*, 5(1–2):197–210, 1987.

[16] Z. Xu and K. Hwang. Modelling the Communication Overhead: MPI and MPL Performance of the IBM SP2. *IEEE Parallel and Distributed Technology*, 4(1):9–23, 1996.