

Developing Scientific Applications in GLU

Pushpa Rao*
Dartmouth College
pushpa@griggs.dartmouth.edu

R. Jagannathan
SRI International
jaggan@csl.sri.com

Abstract

We report on the performance of two scientific GLU applications on a standard workstation network. The two applications we consider are LU decomposition (LUD) and shallow water equations solver both of which are rich in data parallelism. We show that while speedup efficiency is reasonable for small number of workstations, it worsens as the number of workstations increases. We identify the contributing factors, namely, relatively significant non-parallelizable computation and increased communication overhead. We explore ways in which these factors can be overcome for the two applications.

1 Introduction

GLU is a high-level integration language for writing parallel applications. A GLU program is constructed by composing together components written in procedural languages like C or Fortran. A GLU program expresses the parallel structure of the computation, while the computations themselves are expressed in procedural modules. More details of the GLU language can be found in [3, 4].

A simple virtual architecture for GLU that exploits coarse-grain parallelism has been implemented on a network of workstations. The architecture consists of a generator (G) and several executors (E). The generator disseminates pieces of work to the executors in the form of coarse-grain procedural functions that have to be applied to their corresponding argument data. Each executor simultaneously evaluates its ripe procedural function and returns the result to the generator. The distribution of ripe functions to executors is completely dynamic — an executor is memoryless between successive function invocations.

*This work was performed when the author was at SRI International.

```

do k = 1, n - 1
  find pivot p such that  $a_{pk} = \max[a_{kk} \dots a_{nk}]$ 
  interchange rows p and k
   $c = 1 / a_{kk}$ 
  do i = k + 1, n
     $a_{ik} = a_{ik} \times c$ 
  do i = k + 1, n
    do j = k + 1, n
       $a_{ij} = a_{ij} - a_{ik} \times a_{kj}$ 

```

Figure 1: Sequential LUD Algorithm

It has been shown [2] by means of a suite of applications written in GLU that the tournament computation approach can be used effectively to express implicit data-parallelism. In this approach, an aggregate data structure (such as a matrix), is viewed as a collection of sub-structures that could possibly be distributed amongst a number of processors. The computation evaluates sub-results from the sub-structures and combines them in a tree-like fashion and arrives at the final result. Both the applications have been implemented using the tournament computation approach. The resulting programs are simple and expressive.

2 LUD

LUD is an important algorithm which is at the core of many scientific computing applications. The main computation in sequential LUD on a dense matrix of order n is shown in Figure 1.

We first describe a block-oriented LUD algorithm with inherent coarse-grain parallelism. A block-oriented algorithm enables us to express the problem at hand in terms of operations on sub-matrices instead of scalar or vector operations. With a block algorithm, granularity of the operations can be made large and locality of data can be exploited by expressing operations in terms of blocks.

2.1 Block LUD

We implemented the CROUT variant of the block LUD algorithm[1], in which a block row and a block column are computed at each step, using previously computed rows and columns. Every step consists of three computational parts: a sequential unblocked LU factorization within a column block, a set of block

No.of Procs	Total runtime (secs)	Seq. time (secs)	Comm. time (secs)	Proc. Util. %	Speedup Efficiency
1	1050				
2	706.8	184	144.5	58.5	0.74
4	464.0	184	144.5	42.6	0.57
8	359.8	184	141.1	27.4	0.36
1	1050				
2	613	250	60	57	0.85
4	434	250	62	40	0.61
8	384	250	70	23	0.35
1	1050				
2	785.8	301.9	29.1	51.8	0.67
4	643.5	303.2	28.5	32.4	0.41
8	648.7	303	33.9	16.3	0.20

Table 1: Timings: Block LUD for 1024×1024 matrix with block size = 64, 128, 256

matrix multiplications and a sequential inversion of a block triangular matrix. Pairs of blocks are transferred to the workers which compute their product and return the resulting block to the generator. These blocks are then put together to form a new computed block column or block row. The main advantage of this scheme is that the granularity of the remote functions created for remote execution are very uniform. Every remote function consists of multiplying two blocks of data. Hence the blocksize (b) determines the granularity of the remote function. The block algorithm also ensures a high computation-to-communication ratio, since for every $3 \times (b \times b)$ size data being moved between an executor and generator, $b \times b \times b$ work is being done at the executor by the matrix multiplication function.

Table 1 shows the performance of the block LUD algorithm on a matrix of size 1024×1024 matrix with block size 64, 128 and 256 on 1, 2, 4 and 8 executors.

From Table 1, it can be seen that the sequential, unblocked computation dominates the total computation, specially as the number of processors increases. For example, in the case when block size = 128, the sequential computation amounts to 25% of the overall runtime with one executor, it constitutes 41% of runtime with two executors, 58% with four executors, and 65 % with eight executors. Increasing the number of executors does not affect this part.

Another reason for degradation in speedup efficiency with increasing number of executors is that during the course of computation, varying number of parallel remotely executable functions are created as per the algorithm. This number is a function of data-size and block-size only. The number of functions may not be

No.of Procs	Total runtime (secs)	Seq. time (secs)	Comm. time (secs)	Proc. Util. %	Speedup Efficiency
1	1050	250			
2	613	250	60	57	0.85
4	430	106	90	62	0.61
8	383	127	219	37	0.35

Table 2: Timings: Panel LUD for a 1024×1024 matrix.

enough to employ all of the executors available at any given time, consequently reducing overall processor utilization. This phenomenon occurs more frequently when the number of executor increases while the data-size and blocksize remain the same. This is observed, for example, when eight executors are employed with blocksize = 256.

2.2 Panel LUD

This algorithm was chosen to alleviate the shortcomings of the block LUD scheme, namely the sequential bottleneck and the low processor utilization.

The algorithm is an adaptation of the sequential LUD algorithm. Instead of picking a single candidate pivot row in every iteration and performing row-reductions operations on the entire remaining sub-matrix, K candidate rows are picked in every outermost iteration, and K columns of multipliers are computed for subsequent reductions. The remaining sub-matrix is divided into a number of panels and K row-reductions are performed on the panels at available worker-sites. The first panel of the sub-matrix is treated differently from the rest of the panels. It undergoes the K row-reductions corresponding to the current iteration and also produces the next set of candidate rows and multipliers. To maintain balance of load between the first panel and the rest of the panels which undergo only reduction, K is chosen carefully in every iteration so that it is only a fraction of the number of columns in any other panel.

The sequential work of finding the next set of K candidate-rows and multipliers is done concurrently along with reduction of the rest of the panels. Any row permutations that may be done while finding the K candidates are remembered so that the same permutations can be applied to the same rows in all the other panels.

Table 2 shows the performance figures for a size 1024×1024 matrix, with 1, 2, 4 and 8 executors.

It can be seen that the non-parallelizable time has been lowered considerably and the processor utilization has improved especially when the number of executors is 4 or less. However, speedup remains low when the number of executors is eight. This is because, at each step the remaining submatrix is divided

amongst the executors and hence the granularity of the remote functions being created falls rapidly as the algorithm progresses. Even though LUD is quite compute-intensive (1050s of computation for a 1024×1024 matrix), the granularity is not uniformly distributed with Panel LUD. The granularity is quite large at the beginning (averaging over hundreds of seconds per panel) when a much bigger sub-matrix is being reduced, but tapers off drastically (to less than 10 secs per panel) as the sub-matrix being reduced gets smaller. As granularity decreases, executors communicate with the generator at more frequent intervals. With eight executors, the generator-executor communication overhead becomes additive as evidenced by the 219s out of a total runtime of 383s for the eight executor case. (Note that sequential time varies by number of executors because K itself varies.)

The performance of the block LUD scheme shows that scalability is limited by non-parallelizable computation time (as per Amdahl's Law) and even if this time is reduced using a different scheme (panel LUD scheme), coarse granularity has to be maintained to avoid effects of communication overhead.

3 Shallow Water Equations Solver

The shallow water program solves a set of shallow water equations using finite difference approximations on a two dimensional grid. Every iteration of the model requires 65 floating point operations per grid point. Many of these operations use values from immediate neighboring grid points. The GLU program views the grid as a collection of sub-grids. To compute new values for points in a sub-grid it is necessary to have the points that line the sub-grid on all four sides. The granularity can be increased by making it possible for every sub-grid to have n lines of points that are closest to the sub-grid on all sides. This way n iterations can be done on every sub-grid. These n lines are referred to as a *band*.

For our experiments we used a 512×512 grid, with varying sub-grid sizes and band sizes. The total number of iterations was 120. The sub-grids are sent off to the executors along with bands of neighborhood information. At the executor, the sub-grids are subject to as many iterations of finite difference approximations as there are lines of points in the band, and the resulting sub-grid is returned to the generator. The generator puts the sub-grids together into a new 512×512 grid.

In the algorithm we have implemented, the bandsize controls the grainsize. The bigger the bandsize, the more the amount of neighborhood information each sub-grid has, and hence more the number of iterations of finite difference approximations can be performed on the sub-grid. Also larger the bandsize, less the number of times sub-grids have to be sent back and forth between the generator and the executors which reduces the communication overhead. However, the amount of redundant work done per sub-grid increases. This is

grid size	sub-grid size	band size	Num of procs.	Total time (secs)	Comm. time (secs)	Proc util %	Job length (secs)	Speedup Efficiency
512			1	2169				1.0
512	128	10	4	985	262.7	75.4	61.2	0.55
512	128	15	4	938.4	266.7	74	86.8	0.58
512	128	20	4	959.5	204.9	80	127.9	0.56
512	128	30	4	907.8	151.2	83.4	189.4	0.6
512	128	40	4	1069.5	125	85.8	305.8	0.50
512	64	10	8	594.8	304.8	65.4	32.4	0.46
512	64	15	8	597.5	289.4	65.7	49	0.45
512	64	20	8	625.0	181.2	75.5	78.6	0.43
512	64	30	8	662.1	186.9	72.2	119.5	0.41
512	64	40	8	731.8	121.0	82	200.0	0.37

Table 3: Timings : Shallow Water Program with 512×512 grid; panel size = 128, 64; number of iterations = 120

because new values have to be computed for all the relevant points in the band of a sub-grid in each iteration in order to compute the correct new values for points inside the sub-grid. The bigger the band, the more the number of points in the band for which new values need be computed.

We used a 512×512 matrix for our experiments. Instead of square sub-grids we opted to use rectangular row-panels of size 128 and 64. We conducted our experiments with 4 and 8 executors on the network, with bandsize of 10, 15, 20, 30 and 40. To minimise communication, as many row-panels are created as there are executors.

From Table 3 we see that for a given panel (sub-grid) size, when the bandsize is low i.e, 10, 15, communication overheads are relatively high and granularity (job-length) is low because of less number of iterations and less redundant computation being done in the bands. When bandsize is high i.e, 20, 30, 40, etc. the communication costs are much lower but since the amount of redundant computation and the amount of useful computation being done at each sub-grid is higher, granularity is very high. As a result of this trade-off between communication time and the amount of redundant computation for each sub-grid we can see from Table 3 that varying the bandsize has an insignificant effect on the total run time of the program.

The algorithm does not scale well. The reason for this is that for a given grid size, the more the number of sub-grids it is divided into, the more the amount of redundant computation that is performed. In particular, for a given bandsize, twice as much redundant computation is done when 8 executors are employed as compared to when 4 executors are employed, because twice as many sub-grids (row-panels in our case) are created.

4 Conclusions

LU decomposition and Shallow Water Equation solver were implemented in GLU using the tournament computation approach. Their performance was studied on a standard network of SunSparc2 workstations. LUD was implemented using a block-oriented and a panel oriented method.

For the block LUD algorithm, it was seen that there was no straight-forward way to choose an optimal blocksize, since bigger block size meant less communication overhead but more load imbalance amongst the executors and smaller block size meant better load balance but increased communication overhead. The non-parallelizable part of the computation became prominent as the number of executors increased (in accordance with Amdahl's law) and limited the speedup efficiency.

The panel LUD algorithm reduced the amount of non-parallelizable computation considerably. But it was seen that granularity could not be maintained uniformly throughout the algorithm. It was large to begin with but reduced drastically towards the end so that the communication between the generator and executors became frequent. This increased the communication overhead drastically and limited the speedup efficiency.

The shallow water equation solver had a uniform computational structure. Also we could increase the granularity and keep communication overhead low, without affecting the load-balance, by increasing the band size. The amount of redundant computation increased with increase in bandsize and with increase in the number of executors. Hence we could get only moderate speedup for a small number of executors (upto four) and low speedup efficiency for larger number of executors.

To achieve scalable performance for scientific applications with nominal non-parallelizable computation, the communication overhead has to be reduced. For the current GLU implementation (one generator and several executors) on a workstation network to offer scalable performance beyond a handful of workstations for such applications, the load balancing strategy has to be revised. Instead of using dynamic load distribution which requires considerable data movement between the generator and executors and vice-versa, a static load distribution strategy could be adopted. With this strategy, specific remote function invocations would occur on specific executors (which would retain data across invocations). This would significantly reduce communications between the generator and executors at the expense of some load imbalance.

References

- [1] Jack J. Dongarra, Iain S. Duff, Danny C. Sorenson & Henk van der Vorst, "Solving Linear Systems on Vector and Shared Memory Computers," *SIAM* (1991).

- [2] R. Jagannathan & A.A. Faustini, "Tournament Computations in GLU", Proceedings of ISLIP 91, Menlo Park, CA.
- [3] R. Jagannathan & A.A. Faustini, "GLU: A Hybrid Language for Parallel Applications Programming", Technical Report SRI-CSL-92-13, Computer Science Lab, SRI International, Menlo Park, CA 94025, USA, 1992.
- [4] R. Jagannathan & C. Dodd, "The GLU Programmers Guide (version 0.9)", Technical Report SRI-CSL-94-06, Computer Science Lab, SRI International, Menlo Park, CA 94025, USA, 1994.
- [5] Nenad Nedeljkovic & Michael Quinn, "Data Parallel Programming on a network of Heterogenous Workstations", First Intl. Symp. on HPDC, Sep 92.