Challenges in Obtaining Peak Parallel Performance with a Convex C240, a Parallel Vector Processor

Patrick McGehearty
Convex Computer Corporation
3000 Waterview Pkwy.
PO Box 833851
Richardson, TX 75083-3851
{uiucdcs,sun,uunet,harvard,killer,usenix}!convex!patrick
(214) 497-4000

May 1, 1989

Abstract

The behavior of the Linpack 300x300 benchmark is examined in the context of a parallel vector machine architecture. Detailed evaluation is performed with respect to the Convex C240. Issues relating to algorithm design and system characteristics are discussed in the context of the Linpack implementation.

Keywords: algorithms, vector, parallel, linpack, performance, measurement

Overview

This report examines the behavior of the Linpack 300x300 benchmark [Dongarra1] on a parallel vector machine. It is observed that the performance of several parallel vector machines on this application is far below their nominal peak performance. Dissection of the internals of the algorithms shows how peak performance is limited. The insights gained provide guidance to algorithm developers as to ways to make maximum use of architectural strength. Also, system architects may gain insight as to which system characteristics to optimize in order to increase the average performance of their systems for this class of application.

The Linpack 300x300 Benchmark

The Linpack 300x300 benchmark consists of solving a 300x300 matrix using LU factorization. The primary activity of the program involves many separate matrix-vector product computations, with dimensions ranging from 1 to 300 for the vectors and matrices. Other, more complex algorithms for solving a 300x300 matrix have been developed especially for vector architectures to improve machine performance. Those algorithms are not considered here since the primary focus of this study is architectural characteristics rather than the specifics of the problem chosen for study.

This benchmark is interesting to study for several reasons. Due to its free distribution by Dongarra and frequent use in comparing the performance of machines intended for the scientific marketplace, its behavior has been measured on a variety of parallel-vector architectures. It is of a size to demonstrate good performance on a vector machine, typically several times that of the Linpack 100x100 benchmark also distributed by Dongarra. A much higher percentage of peak machine performance is obtained on single vector processors than on parallel-vector processors.

Table 1 compares the performance of several different machines with parallel vector architectures. In all cases, the results show the performance of the same algorithm for solving a 300x300 matrix using LU factorization. The critical matrix-vector computations are coded in assembly language for all machines displayed. Using assembly language results removes differences caused by

different qualities of vectorizing and parallelizing compilers, focusing on basic architectural characteristics. Note that the Convex C2 series architecture achieves the highest percentage of peak performance of any of the architectures shown, for any number of processors. Thus, it is an appropriate vehicle for understanding the interrelationship of parallel vector architectures and the Linpack 300x300 algorithm. While the different architectures obtain a varying degree of success in approaching peak floating point performance with a single vector processor, all architectures obtain less than one half of peak performance when executing on four processors.

Table 1. Percentage of Peak Performance Obtained

machine	Mflops			Percent of Peak		
	1 Proc	2 Proc	4 Proc	1 Proc	2 Proc	4 Proc
Cray Y-MP	250	400	563	0.75	0.60	0.42
Cray X-MP	187	314	459	0.77	0.65	0.47
Cray 2S	270	366	455	0.55	0.38	0.24
Convex C2 series	39	68	96	0.78	0.68	0.48
Alliant FX/80	7	12	21	0.60	0.51	0.45

This sensitivity to parallelism suggests that detailed study will provide useful insights with respect to what factors contribute to limiting parallel vector performance. Identifying these factors so they may be avoided or minimized is likely to be useful to math library and compiler implementors, algorithm designers, and architectures for high performance systems.

The challenge to obtain maximum speed on a parallel-vector computer for this benchmark is large. The primary task is to obtain efficient vector lengths while splitting the work to be done among four processors. A secondary task is to obtain effective speedup from short parcels of work, some less than 450 operations in length when executed serially. These challenges must be successfully answered to obtain maximum performance.

Parallel Execution Characteristics on a Convex C240

A Convex C240 includes four vector processors. Each processor may execute independent streams or "threads" of instructions. Each processor contains its own vector unit that may request a new data value from memory every clock cycle. The basic machine clock cycle is 40 nanoseconds, with the vector unit able to "pipeline" or overlap vector load/store, add/sub, and multiply/divide operations with each other and with scalar operations. Thus, each processor may request up to 200 Mbytes/second of data from the memory system while processing a peak of 50 Mflops/second. The memory system consists of up to 32 banks of memory, with a single bank capable of delivering a 32 bit value every 4 clock periods. When a single processor is executing in 64 bit vector mode, 8 memory banks are kept busy at any given time. The current maximum memory system configuration may deliver a peak of 800 Mbytes/second of data to the processors and I/O system.

The primary method for obtaining coordinated parallel execution for the C240 system involves the use of the *spawn* and *join* instructions. When a thread on any processor executes a *spawn* instruction, a notice is posted by the hardware that indicates that other processors are welcome to assist in parallel execution of a body of work. Any "idle" processors accept the spawn and start execution of a thread at the indicated instruction. Appropriate information must be set up in communication registers before the spawn to allow each thread to determine what portion of the work it should execute, and to establlish a method for detecting when all the work is completed. When properly coded, the number of processors executing the parallel code section does not affect the answer obtained, just the time to obtain it. All timings in this discussion are based on the ideal case where four processors always are available for executing parallel regions of code. Each thread generated by a spawn eventually determines that no more work is available to be done. At that point, the thread executes a *join* instruction. If other threads are still completing their portion of the work, then the processor executing the join instruction either starts working on some other task's parallel activity, or if none is available, goes idle. When the last thread executes the join instruction, it continues execution of the serial portion of the code. In this way, spawn-join implement the "barrier" concept of parallel synchronization. More information about parallel execution on a C240 may be found in [CONVEX] and in [McGehearty]

The first idle processor can attach to a spawn in less than three microseconds, but it can take as long as seven microseconds for all four processors to be working on the same task from a single spawn. The final join instruction requires another microsecond. Therefore, the hardware overhead for a four-way spawn-join is eight microseconds.

When four processors are all accessing memory in vector mode, there is an exact balance between the memory bank supply rate and the processor demand rate. During each vector operation startup, some memory bank accesses collide with other memory banks are idle. After a short period of contention, all four processors get into a lock step access pattern, each being eight banks behind the previous, so that no more memory contention occurs until a vector operation completes. The memory contention due to four processors executing vector/memory operations typically causes a performance loss of approximately 10%.

The code required for standard non-parallel loop setup plus loop termination is approximately ten microseconds. In addition, executing a vector loop in parallel requires more pre-loop scalar computation of loop limits plus set up of values to be communicated to the parallel threads. This time is typically around six microseconds for simple loops. These phases are not parallelizable.

The potential speed of running a given loop which in parallel on four processors can be represented by the formula:

approximate time 4 processors = 8 + 9 + 6 + 1.1*(n-9)/4

where

n microseconds for non-parallel execution

8 microseconds spawn-join overhead for 4 processors

9 microseconds standard loop setup time (non-parallel)

6 microseconds additional parallel loop setup time

1.1 reflects the loss to memory contention

n-9 microseconds for loop portion on single processor

divide by 4 for four processors.

The crossover point, where a benefit from running in parallel is available for loop times of greater than 30 microseconds or approximately six vector operations of length 128 (maximum vector length). The benefit is minimal until substantially longer loops are to be executed in parallel. For example, a sequential loop of length 60 microseconds requires 37 microseconds to execute on four processors.

Linpack Algorithm Phases

The 300x300 Linpack program consists of LU factorization phase followed by a solution phase. The LU phase iterates over five subphases for each row and column, for a total of 300 iterations. The solution phase iterates over the L and U portions of the matrix for a total of 600 separate actions. These subphases are described in Table 2.

Table 2. Algorithm Phases

name	description	purpose	operation count
dmxpy	matrix-vector product	forms Jth column of L	2*(301-J)*(J-1)
idamax	find column maximum	search for next pivot	(301-J)
dswap	interchange two rows	numerical stability	2*300
dxmpy	vector-matrix product	forms Jth row of U	2*(300-J)*(J-1)
dscal	scalar-vector product	normalize columns of L	(301-J)
lus	L + U solution phase	complete solution	2*(301-J) + 2*(J-1)

While the number of operations in each phase varies with J, the average and total may be computed. The solution phase also contains two internal iterations. Table 3 shows the operation count for each phase as well as the execution time. As can be readily seen, most of the work to be done is in the dmxpy and dxmpy phases of computation, also known as the matrix-vector (MV) routines. It is worth noting that the Dongarra benchmark program only counts the floating point operations in the dmxpy and dxmpy routines in computing Mflop rates, omitting the operations required by the other phases. Thus, all machines show an apparent loss of 2% efficiency due to the measurement technique.

Table 3. Operation Counts by Phase

name	operation count		
	on iteration J	total	average/iteration
dmxpy	2*(301-J)*(J-1)	8,999,900	30,000
idamax	(301-J)	45,150	150
swprow	2*300	180,000	1,200
dxmpy	2*(300-J)*(J-1)	8,910,200	29,700
dscal	(301-J)	45,150	150
lus	2*(301-J) + 2*(J-1)	183,000	300
Sum		18,363,400	
Mflop Count		17,910,100	

Analysis of Linpack Short Phases

The short phases are idamax, dswap, dscal, and lus. The idamax phase is implemented by a call to the Convex VECLIB (tm) package which provides a vectorized method for finding the maximum element in a vector. The dswap, dscal, and lus phases use the Convex fc5.0 compiler to generate their code. The matrix-vector routines, dmxpy and dxmpy, where most execution time is spent, are implemented in assembly language.

The three phases idamax, dscal, and lus are judged to be too small for the typical case (< 30 microseconds) to obtain benefit from running in parallel. Some speedup is obtainable for dswap (60 microseconds per iteration is reduced to 37 microseconds), as shown in Table 4. Table 5 summarizes the time required for all the minor phases.

Table 4. Analysis of dswap

sequential	parallel (proc=4)	
51 microseconds	13 microseconds	vector swap time
	+ 1 microseconds	loss to memory contention
	+ 8 microseconds	spawn-join time
	+ 6 microseconds	parallel loop setup
9 microseconds	+ 9 microseconds	standard setup
60 microseconds	37 microseconds	

Table 5. Total of All Minor Phases

	sequential	parallel (proc=4)
idamax	9 milliseconds	9 milliseconds
dswap	18 milliseconds	11 milliseconds
dscal	6 milliseconds	6 milliseconds
lus	10 milliseconds	10 milliseconds
minor sum	43 milliseconds	36 milliseconds

Matrix-Vector Computation

This section discusses the performance of the dmxpy and dxmpy phases. The FORTRAN source code is shown in the Appendix. As there are two loops, either may be used for vectorization. Vectorization on the J = 1, N2 loop would involve sum-reduction, which is inherently a less efficient operation than vector addition, because of the delay required in using the immediately formed sum in computing further results. On the C2 series, sum-reduction has an instruction execution time of VL+43 clocks where a vector to vector add or multiply requires VL+10 clocks (VL = vector length).

Given the choice to vectorize on the I = 1, N1 loop, there are two ways to parallelize this code. Either the J = 1, N2 loop may be broken into portions, or the I = 1, N1 loop may be subdivided into sections. Running the J = 1, N2 loop in parallel has the disadvantage of requiring the final summation of Y to be synchronized with corresponding extra overhead. This method will be called the parallel accumulation method. Running the I = 1, N1 loop in parallel avoids the synchronization step, but yields shorter vector lengths for each processor. This method will be called the split vector method.

A full analysis of what method yields best results requires an analysis at the assembly language level. The Appendix also lists the assembly required for the inner loop of the parallel section. For long vectors, most of the time required for the scalar operations in the loop is overlapped with the vector operations. If the vector length is greater than 44, then the time required for each loop execution is (10 + VL) clock ticks. The maximum vector length a C2 series machine is 128. However, when vector length is short (less than 44), vector and scalar operations are no longer completely overlapped. The number of clocks required to execute the dmxpy loop is max(vl+k,32) where k is either 6 or 10, depending on memory configuration. If there is more memory bandwidth available than executing processors, then the load scalar operation (ld.w (a1),s0) does not create memory bank contention with the load vector operation. Then, k equals 6 clock cycles. If memory bandwidth is matched by processor demand, as when all four processors are executing vector loads and stores, then k equals 10 clock cycles on average. The additional time is required because the scalar reference does not occur in the same pattern as the vector references, so it frequently creates memory bank conflicts with the other processors. With this timing information, the number of cycles required for execution of the inner loop for each of the two parallel methods may be computed.

Split Vector = max(N1/4+10,32) * N2

Parallel Accumulation = max (N1+10,32) * N2 /4 + C * 3 * N1

where C = factor for lock contention on summation step

If N1 >> N2, the Split Vector method is best. If N1 << N2, the Parallel Accumulation method is best. Timing comparisons of both methods indicate that for the 300x300 problem, the crossover point is when N1 is in the range between 100 and 150 (at least when N2 = 300 - N1). Therefore, to combine performance optimization with ease of coding, a transition point of 128 was chosen. The assembly code for dmxpy and dxmpy makes a selection at run time to use the Parallel Accumulation method when N1 <= 128, and the Split vector method when N1 > 128. The use of 128 as the breakpoint made the coding of the Parallel Accumulation method simpler since all temporary results could be retained in a single vector accumulator, and no additional loop overhead was needed in the Parallel Accumulation phase for vector loop strip mining.

Matrix-Vector Measurements

The predicted and actual execution time of various phases of dmxpy under different assumptions are shown in Table 6. The base prediction of 174 microseconds is based on an assumption of linear speedup of the timings of the same routine written for a single processor C210. The average execution time for dmxpy on a single processor is 696 microseconds. If perfect parallelism were achieved, then dmxpy would take 174 microseconds. The table also includes the known architectural and algorithmic characteristics. The additional time for the shorter vector lengths in the Split Vector case or synchronized update in the Parallel Accumulation case averages 12 microseconds. Models of memory contention when all four processors are in pure vector mode indicate a loss of 10% performance. The additional contention caused by the scalar access cost an additional 24 microseconds. The spawn-join time is 8 microseconds, and the additional cost for parallel loop setup is 6 microseconds. The common setup cost is 9 microseconds.

Table 6. Parallel dmxpy Analysis

Feature	Prediction	
matrix-vector product	174	microseconds
adjust for shorter vectors	+ 12	microseconds
loss to vector memory contention	+ 18	microseconds
vector/scalar contention	+ 24	microseconds
spawn-join time	+ 8	microseconds
parallel loop setup	+ 6	microseconds
standard setup	+ 9	microseconds
sum	251	microseconds

Complete Phase Timings

Since dmxpy is invoked 300 times, the architectural and algorithm model predicts a total of 75 milliseconds of execution time required for the dmxpy phase of computation. A similar analysis yields 74 milliseconds as the predicted time for the dxmpy phase of computation. The timings for the other phases shown in Table 7 are based on program measurements. The first column shows the timings for a single processor solution. The second column indicates the predicted results. The third column shows the measured results with well tuned code.

Table 7. Summary and Phase Timings (in milliseconds)

name	C210		C240 (predicted)		C240 (actual)	
	total	average	total	average	total	average
dmxpy	209	0.696	75	0.251	76	0.254
idamax	9	0.030	9	0.030	9	0.030
dswap	18	0.060	11	0.060	11	0.037
dxmpy	207	0.690	74	0.247	75	0.250
dscal	6	0.020	6	0.020	6	0.020
lus	10		10		10	
Sum	459	msec	185	msec	187	msec
Mflops	39	Mflops	97	Mflops	96	Mflops

The observed results column of Table 7 shows a close correlation with the predicted results. Allowing for measurement errors in determining each overhead, a measured results within 1% of the predicted result indicates that the critical factors affecting performance have been accounted for.

Conclusions

From an algorithmic point of view, there are three major limits. These are the non-parallel portions of the program, the problem of short vectors, and the problem of frequent transitions from parallel to sequential mode and back. The problem of short vectors is partially handled by dynamic algorithm selection. Current state of the art compiler technology has been demonstrated that can generate vector and non-vector versions of a loop with the selection of which is to be executed depending on the runtime iteration count [CNXFTN]. Extending such technology to the parallel case is straightforward. Users can aid compilers with more is known about the characteristics of the application and expected iteration counts.

The problem of frequent transitions from parallel to sequential mode and the many short non-parallel portions of the program require more extensive modifications in approach. The non-parallel portions are limited primarily by their size. These issues are less for larger problem sizes, such as solving a 1000x1000 matrix, but are still significant. Alternate algorithms such as those described in [Dongarra2]. have been developed which greatly increase the amount of work done in each parallel phase, reducing the sequential phases to a minimum, and reducing memory bandwidth requirements as well. These methods can increase performance by 70% [Dodson].

In the area of architecture guidance, two issues are clear, memory contention and parallelization costs. In the area of memory contention, providing excess memory bandwidth can reduce memory contention to the point where it is negligable. Improving this feature is not trivial, as memory subsystem costs are a significant fraction of system cost for supercomputers, especially those with large memories. The cost of fast memory components can be dramatically higher than whatever is the current mass market memory. The other common approach to improving memory bandwidth is to increase the number of memory banks. This approach has the cost of increased memory control logic. Innovative solutions in the area of increasing effective memory bandwidth to vector processors would contribute to obtaining peak parallel-vector performance.

The cost of parallelization is a critical design criteria for parallel algorithms. In the C2 series, substantial architectural support is provided to minimize these costs. Special registers are provided for interprocessor communication and synchronization. Hardware instructions allow thread initiation and termination without intervention by the operating system. These features make it feasible to consider allocating and deallocating processors for code segments as short as 6 vector operations. Even so. parallelization costs show up as a critical issue. Future designers need to continue to expend design resources on minimizing these costs.

Appendix: Source code for matrix-vector product subroutines

```
C  ** N1 = 301-J, N2 = J-1 for DMXPY1

      SUBROUTINE DMXPY1 (N1,Y,N2,LDM,X,M)
C ** DMXPY1.F -- DMXPY UNROLLED TO A DEPTH OF 1
C
      INTEGER LDM,N1,N2
      double precision Y(N1),X(N2),M(LDM,N2)
C
      DO 20 J = 1, N2
        DO 10 I = 1, N1
          Y(I) = (Y(I)) + X(J)*M(I,J)
 10     CONTINUE
 20   CONTINUE
      RETURN
      END

C  ** N1 = 300-J, N2 = J-1 for DXMPY1
      SUBROUTINE DXMPY1(N1,LDY,Y,N2,LDX,X,LDM,M)
C ** DXMPY1.F -- DXMPY UNROLLED TO A DEPTH OF 1
C
      INTEGER N1,LDY,N2,LDX,LDM
      double precision Y(LDY,N1),X(LDX,N2),M(LDM,N1)
C
      DO 20 J = 1, N2
        DO 10 I = 1, N1
          Y(1,I) = (Y(1,I)) + X(1,J)*M(J,I)
 10     CONTINUE
 20   CONTINUE
      RETURN
      END

;  Assembly language for inner loop of dmxpy1
;
; on loop entry:
;     a5 = addr M(I,J)
;     a1 = addr X(J)
;     a3 = addr increment for M(I,J)
Lf:
        ld.d      8(a1),s1    ; load X(J)
        add.w     a3,a5       ; increment addr M(I,J)
        add.w     #8,a1               ; increment addr X(J)
        ld.d      0(a5),v0    ; load M(I,J) (for vector length elements)
        mul.d     v0,s1,v1    ; compute X(J)*M(I,J)
        add.d     v2,v1,v2    ; Y(I) = Y(I) + X(J)*M(I,J)
        lt.w      a1,a4       ; last X(J) done?
        jbra.t    Lf          ; if not branch back
```

Bibliography

[CONVEX] **Convex Architecture Reference Manual**, 3rd Edition, October, 1988. Convex Computer Corporation.

[CNXFTN] **Convex Fortran User's Guide**, 8th Edition, October, 1988. Convex Computer Corporation.

, or suitable Convex article.

[Dodson] Dodson, David, personal communication concerning Convex VECLIB performance.

[Dongarra1] Dongarra, Jack J. *Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment.* Argonne National Laboratory, Mathematics and Computer Science Division, Technical Memorandum No. 23. October, 1988.

[Dongarra2] Dongarra Jack J. and Hewitt Tom: Implementing Dense Linear Algebra Algorithms Using Multitasking on the CRAY X-MP-4 (or Approaching the Gigaflop), 347.

[Dongarra3] Dongarra, Jack J. and Stanley C. Eisenstat, *Squeezing the most out of an algorithm in CRAY Fortran*, ACM Trans. Math. Software, 10 (1984), pp. 221-230.

[McGehearty] McGehearty, Patrick. *Parallel Programming in Convex Assembly Language*, Application Note, Convex Computer Corporation.