

The Cedar System and an Initial Performance Study*

D. Kuck, E. Davidson[†], D. Lawrie[†], A. Sameh
C.-Q. Zhu[†], A. Veidenbaum, J. Konicek, P. Yew,
K. Gallivan, W. Jalby[†], H. Wijshoff[†], R. Bramley[†], U.M. Yang
P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon[†], Z. Li[†]
T. Murphy, J. Andrews, S. Turner

Center for Supercomputing Research and Development
University of Illinois
Urbana, IL, 61801

Abstract

In this paper, we give an overview of the Cedar multiprocessor and present recent performance results. These include the performance of some computational kernels and the Perfect Benchmarks[®]. We also present a methodology for judging parallel system performance and apply this methodology to Cedar, Cray YMP-8, and Thinking Machines CM-5.

1 Introduction

Tremendous progress in VLSI technology today has made it possible to build large-scale parallel systems with dazzling peak performances. Several such systems have even been commercialized over the last 5 years. However, the goal of building a general-purpose large-scale parallel system remains quite elusive. Many of the systems still have very little software support and are very difficult to program. The sustainable performance from various real applications running on those machines remains erratic and unpredictable. These phenomena show that we still do not know how to build parallel machines, how to program such machines, or how to characterize the performance of such machines for writing good application codes. The difficulties stem from the fact that in order to build a large-scale parallel machines that can deliver

“practical parallelism,” we need to understand the interactions among system architecture, system software and parallel application codes.

The Cedar project brought together a group of people in the areas of computer architecture, parallelizing compilers, operating systems, and parallel algorithms/applications to help solve the real problems associated with building a “complete” parallel system, and to study the effects of interaction among these components on such a machine [GKLS83, KDLS86]. The machine has been in full operation since late 1990. The Cedar experience includes the architecture, compiler, OS, and application perspectives and this paper attempts to summarize these for the architecture community.

We describe the machine organization in Section 2, concentrating on the unique aspects of Cedar. Programming and compilation for Cedar are discussed in Section 3. Performance measurements of the systems and the interpretation of those results are presented in Section 4.

2 The Organization of Cedar

Cedar is a cluster-based shared memory multiprocessor. The system consists of four clusters connected through two unidirectional interconnection networks to a globally shared memory (Fig. 1). Each cluster is a slightly modified Alliant FX/8 system with eight processors. In this section we first summarize the features of these clusters and then describe the unique features of Cedar. For a more detailed overall description of Cedar see [KTVZ91].

*This research was supported by the Department of Energy under Grant No. DE-FG02-85ER25001 and by the National Science Foundation under Grants No. US NSF-MIP-8410110 and NSF-MIP-89-20891, IBM Corporation, and the State of Illinois.

[†]Affiliated with CSRD for a portion of the project.

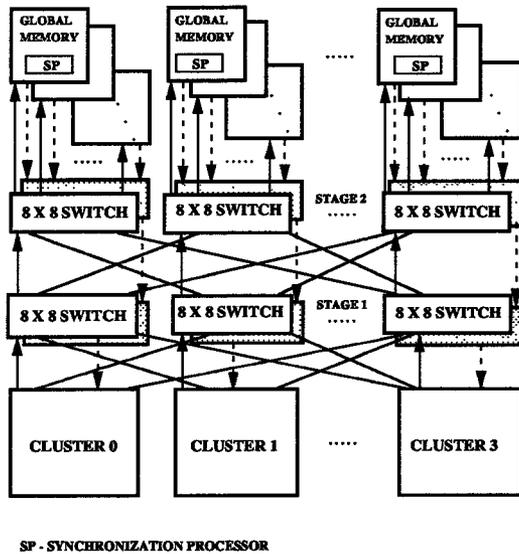


Figure 1: Cedar Architecture

Alliant clusters The organization of an Alliant FX/8 cluster is shown in Figure 2. Each Alliant FX/8 contains 8 *computational elements* (CEs). The CEs are connected to a 4-way interleaved *shared cache* which in turn is connected to an interleaved cluster memory. The FX/8 also includes *interactive processors* (IPs) and IP caches. IPs perform input/output and various other tasks.

The CE is a pipelined implementation of the 68020 instruction set augmented with vector instructions. The CE instruction cycle is 170ns. The vector unit implements 64-bit floating-point as well as integer operations. Vector instructions can have a register-memory format with one memory operand. The vector unit contains eight 32-word registers. The peak performance of each CE is 11.8 Mflops on 64-bit vector operations.

Each CE is connected to a *concurrency control bus* designed to support efficient execution of parallel loops. Concurrency control instructions implement fast fork, join and synchronization operations. For example: concurrent start is a single instruction that “spreads” the iterations of a parallel loop from one to all the CEs in a cluster by broadcasting the program counter and setting up private, per processor stacks. The whole cluster is thus “gang-scheduled.” CEs within a cluster can then “self-schedule” iterations of the parallel loop among themselves.

Each Alliant FX/8 has 32MB of cluster memory. All references to data in cluster memory first check the 512KB physically addressed shared cache. Cache line

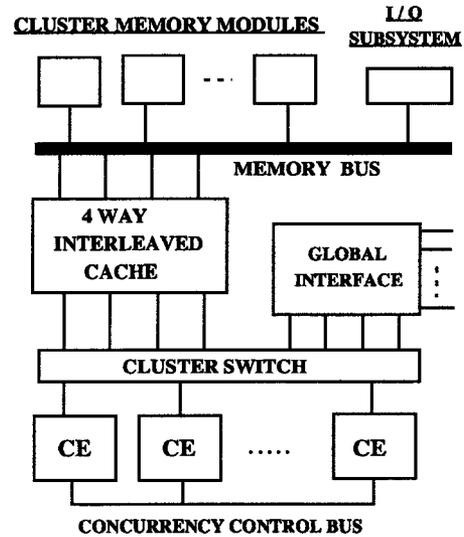


Figure 2: Cluster Architecture

size is 32 bytes. The cache is write-back and lockup-free, allowing each CE to have two outstanding cache misses. Writes do not stall a CE. The cache bandwidth is eight 64-bit words per instruction cycle, sufficient to supply one input stream to a vector instruction in each processor. This equals 48MB/sec per processor or 384MB/sec per cluster. The cluster memory bandwidth is half of that or 192MB/sec.

Memory Hierarchy The Cedar memory hierarchy consists of 64MB of shared global memory and four cluster memories with caches. It supports a virtual memory system with a 4KB page size. The physical address space is divided into two equal halves: cluster memory is in the lower half and shared memory is in the upper half. Global memory is directly addressable and shared by all CEs. Cluster memory is only accessible to the CEs within that cluster. Data can be moved between cluster and global shared memory only via explicit moves under software control. It can be said that cluster memories form a distributed memory system in addition to the global shared memory. Coherence between multiple copies of globally shared data residing in cluster memory is maintained in software. The global memory system is weakly ordered. Global memory is double-word (8byte) interleaved and aligned. The peak global memory bandwidth is 768MB/sec or 24MB/sec per processor.

Global Network The Cedar network was designed to support simultaneous vector loads/stores from global memory by all processors. It is a multistage shuffle-exchange network as shown in Fig-

ure 1. The network is self-routing, buffered and packet-switched. Routing is based on the tag control scheme proposed in [Lawr75], and provides a unique path between any pair of input/output ports. Each network packet consists of one to four 64-bit words, the first word containing routing and control information and the memory address. The network is constructed with 8×8 crossbar switches with 64-bit wide data paths. A two word queue is used on each crossbar input and output port and flow control between stages prevents queue overflow. The network bandwidth is 768MB/sec for the entire system or 24MB/sec per processor, which matches the global memory bandwidth.

Data Prefetch The Cedar *data prefetch unit* (PFU) is designed to mask the long global memory latency and to overcome the limit of two outstanding requests per Alliant CE. Each CE has its own individual PFU. Each PFU supports one vector load from global memory. A PFU is “armed” by giving it the length, the stride and the mask of the vector to be fetched. It is then “fired” by the physical address of the first word to be fetched. This starting address can be supplied by a special prefetch instruction or by the first address of a vector load. In the former case the prefetch is completely autonomous and can be overlapped with computation or cluster memory accesses. In the latter case it is only overlapped with the current vector instruction. When a prefetch crosses a page boundary the PFU suspends until the processor supplies the first address in the new page because only physical addresses are available to the PFU. In the absence of page crossings the PFU issues up to 512 requests without pausing. The data returns to a 512-word prefetch buffer which is invalidated when another prefetch is started. It is possible to keep prefetched data in that buffer and reuse it from there. Data can enter the prefetch buffer from the global network out of order due to memory and network conflicts. A full/empty bit per word allows the CE both to access the buffer without waiting for the prefetch to be complete and to get the data in same order as requested.

Memory-based Synchronization requires read-modify-write operations on global memory. However, given multistage interconnection networks it is impossible to provide standard lock cycles and very inefficient to perform multiple memory accesses for synchronization. Cedar implements a set of indivisible synchronization instructions in each memory module. These include Test-And-Set and Cedar synchronization instructions based on [ZhYe87], which are performed by a special processor in each memory module. Cedar synchronization instructions implement Test-

And-Operate, where Test is any relational operation on 32-bit data (e.g. \geq) and Operate is a Read, Write, Add, Subtract, or Logical operation on 32-bit data. Synchronization instructions are accessible from a CE via memory-mapped instructions initiated by a Test-And-Set to a global memory address.

Performance monitoring The Cedar approach to performance monitoring relies on external hardware to collect time-stamped event traces and histograms of various hardware signals. The event tracers can each collect 1M events and the histogrammers have 64K 32-bit counters. These can be cascaded to capture more events. Each of the major Cedar units has several important signals available for external monitoring. Any other accessible hardware signal can also be monitored. Software tools start and stop the experiments and move the data collected by the performance hardware to workstations for analysis. It is also possible to post events to the performance hardware from programs executing on Cedar, which allows software event tracing.

3 Programming Cedar

Cedar programs can be written using the CEDAR FORTRAN language, Xylem assembler or C. All of these make use of the abstractions provided by the Xylem kernel [EABM91] which links the four separate operating systems in Alliant clusters into the Cedar OS. Xylem exports virtual memory, scheduling, and file system services for Cedar.

A program for Cedar can be written using explicit parallelism and memory hierarchy placement directives. Parallelism can be in the form of DOALL loops or concurrent tasks. Alternatively, programs written in Fortran 77 can be translated automatically into CEDAR FORTRAN by the “Parallelizing Restructurer”. The overall compilation and program execution process is simple and identical to the same process on a single-processor workstation. Writing programs that extract all of Cedar performance is a more challenging task.

The CEDAR FORTRAN language is a dialect of FORTRAN77 that includes parallel and vector extensions. CEDAR FORTRAN is translated into ALLIANT FORTRAN and Alliant’s compiler performs code generation. Programming in FORTRAN77 provides a simpler user interface and better portability, although sometimes at the expense of performance.

CEDAR FORTRAN offers an application programmer explicit access to all the key features of the

Cedar system: the memory hierarchy, the prefetching capability from global memory, the global memory synchronization hardware, and cluster features including concurrency control. This access is supported by language extensions and run-time library functions. The CEDAR FORTRAN language extensions and compilation for Cedar are described next. CEDAR FORTRAN is fully described in [Hoef91] and description of the Cedar Compiler project can be found in [EHJL91, EHLP91, EHJP92].

3.1 Fortran Extensions

Parallel Loops A programmer can express parallelism using DOALL loop constructs. A DOALL is a loop in which iterations are independent and therefore can be executed in parallel. Several flavors of DOALLs are provided in order to better control load balancing, data placement, and scheduling overhead. These include CDOALL, SDOALL, and XDOALL which will be described below.

Data Placement and Sharing Data can be placed in either cluster or shared global memory on Cedar. A user can control this using a GLOBAL attribute. Variable placement is in cluster memory by default. A variable can also be declared inside a parallel loop. The loop-local declaration of a variable makes a private copy for each processor which is placed in cluster memory.

3.2 Compilation for Cedar

Parallel Loops XDOALL makes use of all the processors in the machine and schedules each iteration on a processor. The processors get started, terminated, and scheduled through functions of the run-time library. Since these operations work through the global memory there is a typical loop startup latency of 90 μ s and fetching the next iteration takes about 30 μ s. The second type of parallel loop is the SDOALL which schedules each iteration on an entire cluster. Each iteration starts executing on one processor of the cluster. The other processors in the cluster remain idle until a CDOALL is executed within the body of the SDOALL. The CDOALL makes use of the concurrency control bus to schedule loops on all processors in a cluster and can typically start in a few microseconds. The XDOALL has more scheduling flexibility but also higher overhead. An SDOALL/CDOALL nest has a lower scheduling cost due to the use of the concurrency control bus. Both SDOALL and XDOALL loops

can be statically scheduled or self-scheduled via run-time library options. CDOALL loops are used to exploit the faster loop control and shared cluster memory and cache in a cluster. Data that is private to an SDOALL iteration but shared by all cluster processors will be placed in the cluster memory.

Data Prefetching The compiler backend inserts an explicit prefetch instruction, of length 32 words or less, before each vector operation which has a global memory operand. The compiler then attempts to float the prefetch instructions in order to overlap prefetch operations with computation. This rarely succeeds and thus most of the time prefetch is started immediately before the vector instruction. More aggressive methods are being investigated [GoGV90].

Data privatization CEDAR FORTRAN uses data declared local to a loop in place of scalar and array expansion. In all Perfect programs we have found loop-local data placement to be an important factor in reducing data access latencies. In addition to loop-local declaration, data can be localized by partitioning and distributing them to the cluster memories. Subsequent loops can operate on these data by distributing iterations to clusters according to the data partitions. CEDAR FORTRAN supports this by scheduling iterations of successive SDOALLs on the same clusters.

Global Synchronization The Cedar synchronization instructions have been mainly used in the implementation of the runtime library, where they have proven useful to control loop self-scheduling. They are also available to a Fortran programmer via run-time library routines.

3.3 Program Restructuring

The parallelizing compiler project has two parts. In the first phase we retargeted an early copy of KAP restructurer to Cedar (KAP from KAI as released in 1988) and evaluated its performance. In the second phase we searched for restructuring techniques that improved the performance of real application programs significantly. We did these experiments by manually restructuring the suite of Perfect Benchmarks® programs, using techniques that may be automated in an eventual implementation of the parallelizer. The results are summarized in Table 3.

The table lists speed improvements over the serial execution time of two versions of the Perfect programs. The results in column "Compiled by Kap/Cedar" show that with the original compiler most programs have very limited performance improvement. This happened even though we set compiler options accord-

ing to the Perfect Benchmarks rules. Specifically, in a few cases program execution was confined to a single cluster to avoid intercluster overhead.

The *Automatable* column presents the performance of the programs to which we applied compiler transformations by hand. The name automatable is somewhat optimistic because we have not yet implemented these transformations in an actual parallelizer. However we believe that most of the applied transformations are realizable. These transformations include array privatization, parallel reductions, advanced induction variable substitution, runtime data dependence tests, balanced stripmining, and parallelization in the presence of `SAVE` and `RETURN` statements. Many of these transformations require advanced symbolic and interprocedural analysis methods. The transformations have been described in more detail in [EHLP91, EHJL91, EHJP92]

4 Cedar Performance

Examples of Cedar performance are discussed in this section. Given the complexity of the Cedar architecture, compilers, OS, and of the codes themselves it is very difficult to isolate the performance effects of various architectural features at the level of full codes. Therefore, we start by considering data from well-understood algorithms and kernels which are much smaller and can be modified easily to explore the system. We continue with the performance of the Perfect codes achieved via automated and automatable restructuring transformations. Next we comment on the performance improvements possible for some of the codes when algorithmic and architectural knowledge is used to transform the codes in nonautomatable ways. Finally, we present a methodology for judging parallel system performance and apply this methodology to Cedar, Cray YMP-8, and Thinking Machines CM-5.

4.1 Memory System Performance

The effect of the memory system on performance can be demonstrated by considering three versions of a matrix primitive which computes a rank-64 update to an $n \times n$ matrix. For all versions the matrices reside in global memory. The difference between the versions lies in the mode of access of the data and the transfer of subblocks to cluster cache. Specifically: in the GM/no-pref version all vector accesses are to global

memory and do not use prefetching, the GM/pref version is identical with the exception that prefetching is used and the GM/cache version transfers a submatrix to a cached work array in each cluster and all vector accesses are made to the work array. All versions chain two operations per memory request.

The performance difference for a matrix of size $n = 1K$ (see Table 1) between the three versions is solely due to memory latency. The table shows megaflops achieved for the three versions. The performance of the GM/no-pref version is determined by the 13 cycle latency of the global memory and the two outstanding requests allowed per CE, and is typical of codes that cannot effectively exploit prefetching in their global accesses. The aggressive use of prefetch can mitigate this latency effectively for up to 16 CEs. The GM/pref version demonstrates this with performance improvement factors of 3.5 and 2.9 on 8 and 16 CEs respectively. For three and four clusters, the effectiveness of prefetching is reduced and improvements of only 2.2 and 1.9 are observed. To achieve a significant fraction of the 376 MFLOPS absolute peak performance (or the 274 MFLOPS effective peak due to unavoidable vector startup) the caches in each cluster must be used. The GM/cache version achieves improvements over GM/no-pref that range from 3.5 on one cluster (identical to the effect of prefetch) to 3.8 on four clusters. The 32 CE observed performance yields 74 % efficiency compared to the effective peak and is consistent with the observed maximum bandwidth of memory system characterization benchmarks [GJTV91].

	1 cl.	2 cl.	3 cl.	4 cl.
GM/no pref	14.5	29.0	43.0	55.0
GM/pref	50.0	84.0	96.0	104.0
GM/Cache	52.0	104.0	152.0	208.0

Table 1: MFLOPS for rank-64 update on Cedar

To explore the cause of the degradation of the effectiveness of prefetch for more than two clusters we analyze the performance of four computational kernels using the hardware performance monitor. We consider a vector load (VL), a tridiagonal matrix-vector multiply (TM), the rank-64 update of a matrix (RK), and a simple conjugate gradient algorithm (CG) to show directly the behavior of the global memory and networks in Table 2. The codes use 8, 16, and 32 processors, global data only, and prefetching. The RK kernel prefetches blocks of 256 words and aggressively overlaps it with computation, the other codes use compiler-generated 32-word prefetches. The metrics

used are first word Latency and Interarrival time between the remaining words in the block, in instruction cycles. These are measured for every prefetch request by recording when an address from the prefetch unit is issued to the forward network and when each datum returns to the prefetch buffer via the reverse networks from memory. Minimal Latency is 8 cycles and minimal Interarrival time is 1 cycle. The cycles needed to move data between the CE and prefetch buffer complete the 13 cycle latency mentioned above. Monitoring these times required access to internal hardware signals and is not possible on all processors. As a result, we monitored all requests of a single processor and compared repeated experiments for consistency. The results of all experiments were within 10% of each other.

# CEs	Prefetch Speedup			Latency (cycles)			Interarrival (cycles)		
	8	16	32	8	16	32	8	16	32
TM	2.1	2.0	1.5	9.4	10.2	14.2	1.1	1.2	2.1
CG	2.4	2.2	1.5	9.4	10.3	15.1	1.1	1.2	2.1
VF	1.8	1.7	1.5	9.6	11.0	16.7	1.2	1.4	2.2
RK	3.4	2.9	1.8	12.9	15.3	18.3	1.2	1.8	3.2

Table 2: Global memory performance

The results in Table 2 show that global memory degradation due to contention causes the reduction in the effectiveness of prefetching as the number of CEs used increases. For one cluster both the latency and interarrival time are near their minimums. RK degrades most quickly due to the fact that it uses the longest prefetch block and overlaps all operations with memory accesses. VF is also dominated by memory accesses but degrades less quickly due to the smaller prefetch block which reduces access intensity. The TM and CG kernels suffer approximately the same degradation and are affected less than the others due to the presence of register-register vector operations which reduce the demand on the memory system. We have shown via detailed simulations that this degradation is not inherent in the type of network used but is a result of specific implementation constraints [Turn93].

4.2 Cedar Performance Using the Perfect Codes

All the results presented in this section were collected in single-user mode to avoid the non-determinism of multiprogramming. The results are shown in Table 3 and have speed improvements versus uniprocessor scalar versions of the same codes. Comparison of automatic restructuring with automatable transformations was given in Section 3.3.

Effect of Cedar Features. The results for the versions derived from automatable transformations in Table 3 assume the use of compiler-generated prefetch and Cedar synchronization in the runtime library. “Slowdown” with respect to “Automatable” results for some of these codes when Cedar synchronization is not used for loop scheduling is due to parallel loops with relatively small granularity requiring low-overhead self-scheduling support, e.g., DYFESM and OCEAN. “Slowdown” without the use of prefetching (given with respect to “No Synchronization” results) for many of the codes is typically due to one of two reasons. The first is a domination of scalar accesses, e.g., TRACK. The second is the presence of a large amount of short-lived loop-local data that is placed in cluster memory. The automatable version of the code DYFESM benefits significantly from prefetch due to the large number of vector fetches from global memory on a small number of processors (due to the limited parallelism available).

Hand Optimization. It is possible to improve the execution time of the Perfect codes using knowledge of their main algorithms and of Cedar.¹ Table 4 contains the execution times for these updated codes. Some of the codes were analyzed resulting in changes that ranged in complexity from a simple modification of I/O (BDNA) to a complete rewriting (TRFD).

The execution time for BDNA is reduced to 70 secs. by simply replacing formatted with unformatted I/O. Careful consideration of ARC2D reveals a substantial number of unnecessary computations. Primarily due to their elimination but also due to aggressive data distribution into cluster memory the execution time is reduced to 68 secs. [BrBo91]. If a hand-coded parallel random number generator is used, QCD can be improved to yield a speed improvement of 20.8 rather than the 1.8 reported for the automatable code.

FLO52, DYFESM, and TRFD require more elaborate analyses and modifications. Four of the five major

¹We use prefetch but not Cedar synchronization.

Program	Compiled by Kap/Cedar time (Improvement)	Auto. transforms time (Improvement)	W/o Cedar Synchronization time (% slowdown)	W/o prefetch time (% slowdown)	MFLOPS (YMP-8/Cedar)
ADM	689 (1.2)	73 (10.8)	81 (11%)	83 (2%)	6.9 (3.4)
ARC2D	218 (13.5)	141 (20.8)	141 (0%)	157 (11%)	13.1 (34.2)
BDNA	502 (1.9)	111 (8.7)	118 (6%)	122 (3%)	8.2 (18.4)
DYFESM	167 (3.9)	60 (11.0)	67 (12%)	100 (49%)	9.2 (6.5)
FLO52	100 (9.0)	63 (14.3)	64 (1%)	79 (23%)	8.7 (37.8)
MDG	3200 (1.3)	182 (22.7)	202 (11%)	202 (0%)	18.9 (1.1)
MG3D ^a	7929 (1.5)	348 (35.2)	346 (0%)	350 (1%)	31.7 (3.6)
OCEAN	2158 (1.4)	148 (19.8)	174 (18%)	187 (7%)	11.2 (7.4)
QCD	369 (1.1)	239 (1.8)	239 (0%)	246 (3%)	1.1 (11.8)
SPEC77	973 (2.4)	156 (15.2)	156 (0%)	165 (6%)	11.9 (4.8)
SPICE	95.1 (1.02)	NA	NA	NA	0.5 (11.4)
TRACK	126 (1.1)	26 (5.3)	28 (8%)	28 (0%)	3.1 (2.7)
TRFD	273 (3.2)	21 (41.1)	21 (0%)	21 (0%)	20.5 (2.8)

^aThis version of MG3D includes the elimination of file I/O.

Table 3: Cedar execution time, megaflops, and speed improvement for Perfect Benchmarks

Code	Time	Improvement
ARC2D	68	2.1
BDNA	70	1.7
DYFESM	31	2.2
FLO52	33	1.9
SPICE	26	3.7
TRFD	7.5	2.8
QCD	21	11.4

Table 4: Execution times (secs.) for manually altered Perfect Codes and improvement over automatable w/ prefetch and w/o Cedar synchronization

routines in FLO52 require a series of multicluster barriers. Unfortunately, the associated synchronization overhead degrades performance for problems that are not sufficiently large, e.g., the Perfect data set. Analysis of the algorithms reveals that by introducing a small amount of redundancy, we can transform the sequence of multicluster barriers into a single multicluster barrier and four independent sequences of barriers that can exploit the concurrency control hardware in each cluster. This along with eliminating several recurrences in the remaining major routine results in an execution time of 33 secs.

The major problem with DYFESM is the very small problem size used in the benchmark. If some of the data structures are reshaped and certain key kernels are reimplemented, aggressively using Cedar's

prefetch unit via Xylem assembler, the execution time drops to around 40 secs. If we change the algorithm used in the code and exploit the hierarchical SDOALL/CDOALL control structure an execution time of 31 secs. results [YaGa93].

The execution time of TRFD was reduced to 11.5 secs. by implementing high performance kernels to efficiently exploit the clusters' caches and vector registers [AnGa93]. The improved version was shown to have almost four times the number of page faults relative to the one-cluster version and was spending close to 50% of the time in virtual memory activity. The extra faults are TLB miss faults as each additional cluster of a multicluster version first accesses pages for which a valid PTE exists in global memory. Based on analysis of the virtual memory performance of Cedar, [MaEG92], a distributed memory version of the code was developed to mitigate this problem and yielded a final execution time of 7.5 secs.

SPICE also benefits significantly from algorithmic attention. After considering all of the major phases of the application and developing new approaches where needed the time is reduced to approximately 26 secs.

4.3 Judging Parallelism

In this section we present a performance evaluation methodology for parallel systems. We will cast this discussion in general terms and will compare initial Cedar performance data to commercially available

systems. However, our goal is not to argue that system X is better than system Y, but rather to shed light on how to understand performance and thereby to make parallel processing a practical real-world technology in the future. We first define five practical parallelism tests, then some performance metrics, and finally discuss acceptable performance levels.

The Practical Parallelism Tests

Practical parallelism has not yet been demonstrated; in fact, no standard definition of it exists. It seems clear that there should be “laboratory level” and “commercial level” criteria for judging practical parallelism, and we will now propose five criteria that form a Practical Parallelism Test.

At the laboratory level, we will use as our criterion for the success of parallelism,

The Fundamental Principle of Parallel Processing (FPPP): Clock speed is interchangeable with parallelism while (A) maintaining delivered performance, that is (B) stable over a certain class of computations.

There are really three statements in the FPPP: first, the well-established point that high peak speeds are possible through parallelism, and then two important constraints that we shall use as Practical Parallelism Tests (PPT’s) 1 and 2.

Practical Parallelism Test 1. Delivered Performance: The parallel system delivers performance, as measured in speedup or computational rate, for a useful set of codes.

Practical Parallelism Test 2. Stable Performance: The performance demonstrated in Test 1 is within a specified stability range as the computations vary with respect to certain program structures, data structures, and data sizes.

Next we discuss two additional tests that must be met if one has demonstrated the FPPP and wants to use it in a commercially viable product.

Practical Parallelism Test 3. Portability and Programmability: The computer system is easy to port codes to and to program, for many applications.

Practical Parallelism Test 4. Code and Architecture Scalability: The computer system effectively runs each code/data size on a range of processor counts, and each code’s data size can be scaled up or down on a given architecture.

Finally, if the first system is a success and the company is to survive over time, the system must demonstrate:

Practical Parallelism Test 5. Technology and Scalable Reimplementability: The system

architecture must be capable of being reimplemented (scaled up) with much larger processor counts in the current technology or in new, faster or less expensive technologies as they emerge.

In what follows, we will expand these ideas and illustrate methods by which we can observe the PPT’s and track progress toward satisfying them over time. Despite the great enthusiasm for parallel processing today, not even the Fundamental Principle of Parallel Processing has been demonstrated generally. Substantial amounts of work will be required before the remaining three PPT’s are passed.

For at least twenty years we have used speedup and efficiency as abstract measures of performance. In addition, we now define stability, St , on P processors of an ensemble of computations over K codes as follows:

$$St(P, N_i, K, e) = \frac{\min performance(K, e)}{\max performance(K, e)},$$

where N_i is the problem size of i -th code, and e computations are excluded from the ensemble because their results are outliers from the ensemble (and each code may have a different number of data sets). Instability, In , is defined as the inverse of Stability. The traditional megaflops (millions of floating-point operations per second) are used as our rate measure. We avoid debating how to define floating-point operation counts by simply using the floating-point counts obtained from the Cray Hardware Performance Monitor.

Acceptable Performance Levels Given a speedup we are confronted with the question of “how good is good?” We should answer this using some function of the total number of processors P . Most people experienced with running a variety of real codes on parallel machines would be pleased if they could achieve some fraction of P or even $P/\log P$, especially as P grows to hundreds or thousands of processors. Note that in terms of the 10X/7 years performance improvement achieved over the history of supercomputing, which has mainly been due to hardware speed increases, such a 1000 processor machine would provide about 15 equivalent years of electronics-advancement speed improvement. Thus, if we knew how to build machines that met all five of the PPT’s, $O(P/\log P)$ would be a good performance level. For this discussion, we shall use $P/2$ and $P/2 \log P$, for $P \geq 8$, as levels that denote **high performance** and **acceptable performance**, respectively. We refer to speedups in the three bands defined by these two levels as high, intermediate, or unacceptable.

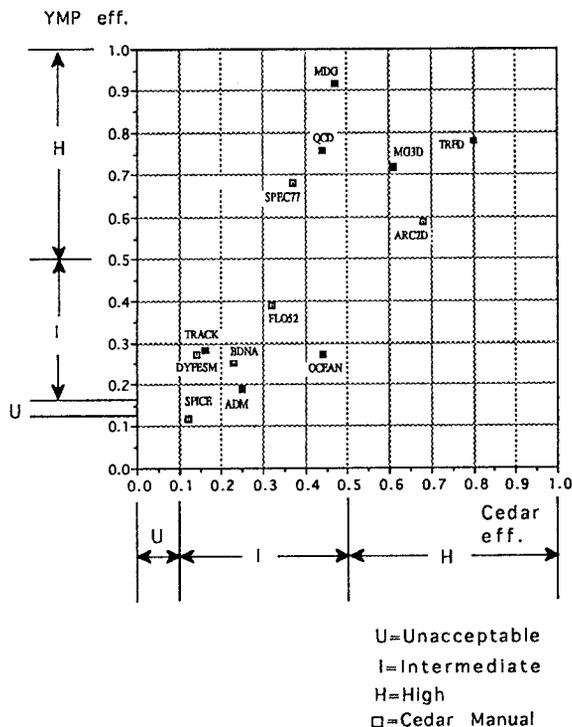


Figure 3: Cray YMP/8 vs. Cedar Efficiency

Applying the Methodology

In this section we apply the ideas of the previous section by using the first four PPT's to analyze Cedar and compare it to other systems.

Practical Parallelism Test 1 is the easiest of the PPT's to meet, and has been demonstrated many times by showing that one code or another runs well on a parallel machine. Figure 3 shows a scatter plot of Cray YMP/8 vs. Cedar efficiencies for the manually optimized Perfect codes. The 8-processor YMP has about half high and half intermediate levels of performance, while the 32-processor Cedar has about one-quarter high and three-quarters intermediate. Note that the YMP has one unacceptable performance, while Cedar has none. We conclude from this that both the Cray YMP and Cedar are on the average acceptable, delivering intermediate parallel performance and thus pass PPT1 for the Perfect codes.

Practical Parallelism Test 2 is much more difficult to meet than PPT1, as it requires that a whole set of programs runs well on some parallel machine. To show the world that it really is a practical system, a parallel machine should be demonstrated on a wide range of program structures, data structures and data sizes. What stability range can one expect on

supercomputers in general, and, at the other extreme, what could one expect on workstations? For the past 20 years, from the VAX 780 through various modern workstations (Sun SPARC2, IBM RS6000), an instability of about 5 has been common for the Perfect benchmarks. Users are evidently not concerned with such computational rate variations on workstations, so we will define a system as **stable** if $\frac{1}{6} \leq St(K, e)$, for small e , and as **unstable**, otherwise. However, if on some architecture, the computational rate varied by substantially more, the net effect could be noticed by many users.

Table 5 shows that Cedar and the Cray YMP/8 both have terrible instabilities for their baseline-automatable computations. This is generally caused by several very poor performers (e.g., SPICE) and several very high performers. So we are led to examining the number of exceptions required to achieve workstation-level stability. We find that two exceptions are sufficient on the Cray 1 and Cedar, whereas the YMP needs six – about half of the Perfect codes. Thus, the YMP cannot be judged as passing PPT2 for the Perfect codes, i.e., it is unstable, while the other two systems do pass with two exceptions.

	$In(13, 0)$	$In(13, 2)$	$In(13, 6)$
Cedar	63.4	5.8	–
Cray 1 ^a	10.9	4.6	–
YMP/8	75.3	29.0	5.3

^awith modern compiler

Table 5: Instability for Perfect codes

It may be regarded as absurd to invoke a test that can be passed by ignoring codes that give top performance. On the other hand, consider two points. First, it can be argued that if a few codes have very high performance, then users should expect that (eventually) all codes will perform well, as is the case with workstations. Secondly, the Perfect codes have relatively small data sizes and stability is a measure that can focus us on the class of codes that are well matched to the system, so varying the data size and observing stability would be instructive. Both of these points indicate that the general purpose abilities of a system are related to its stability.

Practical Parallelism Test 3 Automatic compilation will go a long way towards portability and programmability of parallel systems. Our discussion of PPT3 will thus center on the performance levels that

can be obtained by using compilers on parallel systems. As we discussed earlier, fully automatic compilation is not yet available on Cedar. However, we have developed several compiler enhancements that have not yet been implemented but have been applied manually to the Perfect codes. These “automatable” results are reported together with automatic results for CRAY YMP/8 in Table 6. We conclude from this that the state of today’s compiler art indicates that acceptable levels for the Perfect codes may be reached in the next few years. Thus, we can expect PPT3 to be passed by parallel systems in the near future.

Performance Level	Cedar	Cray YMP
High ($E_p \geq .5$)	1 Codes	0 Codes
Intermediate ($E_p \geq 1/2 \log P$)	9 Codes	6 Codes
Unacceptable ($E_p < 1/2 \log P$)	3 Codes	7 Codes

Table 6: Restructuring Efficiency

Absolute Performance: Cray YMP/8 vs. Cedar The previous discussion has ignored absolute performance in terms of time or megaflops. Table 3 shows the megaflops generated by Cedar automatable versions [Add1], as well as Cray YMP/8 baseline compiler MFLOPS to Cedar MFLOPS ratios. The harmonic mean for the MFLOPS on the YMP/8 is 23.7, 7.4 times that of Cedar. It should be remembered that the ratios of clock speeds of the two systems is $170\text{ns}/6\text{ns} = 28.33$.

Practical Parallelism Test 4 Parallel system performance can vary widely as a function of both the number of processors used and the size of the problems being solved. We shall use the High and Intermediate efficiency (see Table 6) and stability range of $.5 \leq St(P, N, 1, 0) \leq 1$ as acceptability criteria. The system is scalable in a range of processor counts and problem sizes where these criteria are satisfied. We are being more restrictive here than we were in PPT 2. This is reasonable because one should expect less variation in performance when varying data size alone than when varying data size as well as program and data structures, as is done across any benchmark suite. Furthermore, based on a number of measurements (not shown here) an Instability of 2 seems reasonable to expect on workstations as data size varies.

It is difficult to vary the problem sizes and col-

lect scalability information for the Perfect codes, so we drop to the level of algorithm studies here, and use the Thinking Machines, Inc. CM-5 for comparison because it has a larger number of processors than CRAY YMP/8. The performance of a conjugate gradient (CG) iterative linear system solver was measured on Cedar while varying the number of processors from 2 to 32. This computation involves 5-diagonal matrix-vector products as well as vector and reduction operations of size N , $1K \leq N \leq 172K$. Cedar exhibits scalable high performance for matrices larger than something between 10K and 16K, and on up to the largest problems run. Cedar exhibits scalable intermediate performance for smaller matrices, evidently ranging well below the smallest actual runs of $N = 1K$. No unacceptable performance was observed in the data that was gathered.

In [FWPS92], a number of linear algebra experiments are reported on the CM-5. For comparison, we quote data for matrix-vector products with bandwidths 3 and 11. The CM-5 used does not have floating-point accelerators. For problem sizes run, $16K \leq N \leq 256K$, high performance was not achieved relative to 32, 256, or 512 processors. The communication structure of the CM-5 evidently causes these performance difficulties [FWPS92]. The CM-5 exhibits scalable intermediate performance with these three processor counts for problems evidently smaller than 16K for bandwidth 11 and evidently much smaller problems for bandwidth 3. No unacceptable performance was observed in the ranges reported.

Thus we conclude that, for these problems and for the number of processors studied, CM-5 is scalable with intermediate performance, while for up to 32 processors Cedar is scalable with high performance for many problem sizes and with intermediate performance for debugging sized runs.

Absolute Performance: The 32-processor Cedar delivers between 34 and 48 MFLOPS as the CG problem size ranges from 10K to 172K. On the banded matrix-vector product, the 32-processor CM-5 delivers between 28 and 32 MFLOPS for BW=3 and between 58 and 67 MFLOPS for BW=11, as the problem sizes range from 16K to 256K. Thus, the per-processor MFLOPS of the two systems on these problems are roughly equivalent.

We are in the process of collecting detailed simulation data for various computations on scaled-up Cedar-like systems. This takes us into the realm of PPT 5 which we shall not deal with further, in this paper.

Acknowledgments.

We acknowledge the contributions of D. Gajski, R. Downing, T. Tilton, M. Haney, R. McGrath, R. Barton, M. Sharma, D. Lavery, A. Malony, M. Farnwald, S. Midkiff, M. Guzzi, V. Guarna, C. Polychronopoulos, L. Harrison and T. Beck.

References

- [Add1] CSRD Staff. *Perfect Report 2: Addendum 1*. Center for Supercomputing Research and Development, University of Illinois, 1991.
- [AnGa93] J. Andrews and K. Gallivan. *Analysis of a Cedar Implementation of TRFD*, CSRD Report in preparation, University of Illinois.
- [BrBo91] R. Bramley and J. Bordner. *Sequential Optimization and Data Distribution for ARC2D on the Cedar Hierarchical Multiprocessor*, CSRD Report No. 1128, University of Illinois, 1991.
- [EABM91] Emrath, P., et al. *The Xylem Operating System*. Procs. of ICPP'91, vol. 1, pg. 67-70, 1991.
- [EHJL91] Eigenmann, et al. *Restructuring Fortran Programs for Cedar*. Procs. of ICPP'91, vol. 1, pp. 57-66, 1991.
- [EHJP92] Eigenmann, et al. *The Cedar Fortran Project*. CSRD Report No. 1262, University of Illinois, 1992.
- [EHL91] Eigenmann, et al. *Experience in the automatic Parallelization of Four Perfect-Benchmark Programs*. Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, pp. 65-83, 1991.
- [FWPS92] Ferng, W., et al. *Basic Sparse Matrix Computations on Massively Parallel Computers*. AHPCRC Preprint 92-084, University of Minnesota, July, 1992.
- [GJTV91] K. Gallivan, et al. *Preliminary Performance Analysis of the Cedar Multiprocessor Memory System*. Proc. 1991 ICPP, Vol. I, pp. 71-75, 1991.
- [GJWY93] K. Gallivan, et al. *Comments on a Cedar Implementation of FLO52*, CSRD Report in preparation, University of Illinois.
- [GKLS83] Gajski, D., et al. *CEDAR - a Large Scale Multiprocessor*, Procs. 1983 ICPP, pp. 524-529, 1983.
- [GoGV90] Gornish, E., et al. *Compiler-directed Data Prefetching in Multiprocessors with Memory Hierarchies*. Procs. ICS'90, Amsterdam, The Netherlands, vol. 1, pp. 342-353, 1990.
- [Hoef91] Hoeflinger, J. *Cedar Fortran Programmer's Handbook*. Center for Supercomputing Research and Development, University of Illinois, 1991.
- [KDLS86] Kuck, D., et al. *Parallel Supercomputing Today and the Cedar Approach*, Science, vol. 231, pp. 967-974, Feb. 28, 1986.
- [KTVZ91] Konicek, J., et al. *The Organization of the Cedar System*. Procs. ICPP'91, vol. 1, pg. 49-56, 1991.
- [Lawr75] Lawrie, D., *Access and Alignment of Data in an Array Processor*. IEEE Trans. on Computers, vol. C-24, no. 12, pp. 1145-1155, Dec. 1975.
- [MaEG92] B. Marsolf, P. Emrath, and K. Gallivan. *Investigation of the Page Fault Performance of Cedar*, CSRD Report No. 1263, University of Illinois, 1992.
- [Turn93] S. Turner. *Performance Analysis of Interconnection Networks*, PhD Thesis in preparation, 1993.
- [YaGa93] U. M. Yang and K. Gallivan. *Analysis of a Cedar Implementation of DYFESM*, CSRD Report No. 1284, University of Illinois, 1993.
- [ZhYe87] Zhu, C-Q. and Yew, P-C., *A Scheme to Enforce Data Dependence on Large Multiprocessor Systems*. IEEE Transactions on Software Engineering, vol. SE-13, no. 6, pp. 726-739, June 1987.