Cedar Architecture and Its Software

Perry A. Emrath David A. Padua Pen-Chung Yew

Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

The Cedar system is a clustered shared-memory multiprocessor system. It is a general purpose machine designed to support parallel processing for a wide range of numerical and non-numerical applications. A 4-cluster 32-processor prototype is currently being implemented at the University of Illinois at Urbana-Champaign. The architecture and the main system features of the Cedar system were designed to meet the following goals: (1) To be a general-purpose high-performance machine for parallel processing; (2) To be scalable both architecturally and physically to a very large system; (3) To be easy to program and with a good environment for user support.

To achieve those design goals, software on the Cedar system plays a very important role. It reflects the underlying architecture in the language, the compilers and the operating system. The Cedar Fortran language will give the programmer access to some important architectural features of the machine, and in this way facilitate control over the performance of the parallel program. The parallelizing compiler will transform ordinary sequential programs or sequential segments of parallel programs into their parallel equivalents. This will facilitate the programming task. The operating system includes facilities to control the memory hierarchy. It also takes care of scheduling one or more processors for the execution of tasks that may include parallel loop constructs in their code. High system performance can be achieved in Cedar through various levels of parallelism which ranges across algorithms, programs, tasks, etc. In this paper, we will describe the interwoven relations between the software and the underlying hardware, including the architecture, on the Cedar system.

1. Introduction.

One of the most important technological achievements of this century has been the performance gains in digital computers during the past 40 years. The two most important factors in the achievement of these gains have been the progress made in electronic components and in machine organization. As the progress in electronic components slows down, machine designers have to rely more on the organizational aspects to increase performance. At the same time, the decrease in the cost of electronic components makes possible the production of more complex machines.

One architectural approach to increase performance has been the introduction of parallelism inside the CPU by the use of pipelines and multiple functional units. This approach was used in early high-performance machines such as the CDC6600 [Thor70], IBM 360/91 [AnST67], and TI-ASC [Wats72], and is used today in most supercomputers and minisupercomputers such as the Cray systems [Russ78], [Chen84], CDC Cyber 205, Fujitsu VP systems [MiUe84], Hitachi S-810 [NIOK84], NEC SX systems [Wata87], Alliant FX/8, and Convex C-1.

Parallelism can also be introduced at a higher level in the form of multiprocessors. Many of the more recent supercomputers such as the Cray X-MP, Cray 2 and ETA-10, and minisupercomputers such as the Alliant FX/8 and Eissi are multiprocessors. This is a very appealing approach since it can be used not only to accelerate the execution of a single program but also to increase throughput and reliability. Furthermore, the existence of independent control units makes possible the execution of parallel loops with branching, subroutine calls and random memory activity in a more effective way than is possible in single processor vector machines.

2. Organization of the Cedar Multiprocessor

The Cedar supercomputer [KDL86] was designed and is being built at the Center for Supercomputing Research and Development of the University of Illinois. The system consists of multiple processor clusters connected through an interconnection network to a globally shared memory (Fig. 1). The current prototype machine...
Global Switch has 4 clusters, each a slightly modified Alliant FX/8 multiprocessor, and 32 global memory modules. The Cedar prototype system is constructed to facilitate the study of parallel programming techniques and tools, and the effectiveness of parallel processing over a wide range of applications.

Each Cedar cluster may contain up to 8 computation elements (CE) or main processors. These CEs are connected to a shared cache and a concurrency control bus. The cache is connected to the Alliant's main memory through the memory bus. The memory bus has a maximum bandwidth of 188 Mbytes/second. The FX/8 can also include up to 12 interactive processors (IP) to execute the interactive components of user jobs, and to perform input/output and other operating system activities. The IPs are connected to a cache (up to three IPs may share a cache) which in turn is connected to main memory through the memory bus. Coherence between the main memory, CE shared cache and the IP caches is maintained by the memory bus. Between each CE and a global network input port, there is a global interface which performs several important functions. Among them is to provide a pathway between the CE and the global network.

2.1. Computation Elements (CEs)

The Alliant FX/8 CEs are pipelined processors with a 88020-based instruction set augmented with vector instructions. Each CE has a floating-point unit which consists of one adder, two multipliers and one divider. All these floating point units can handle 32-bit and 64-bit floating-point operations. The peak performance of each CE is 6 Mflops (million floating-point operations per second) for 64-bit floating-point vector operations, and 12 Mflops for 32-bit floating-point vector operations. However, there are some vector instructions which allow two vector operations to be "chained" together, e.g. 
\[ c \times B + C \rightarrow D, \]
where `c` is a scalar and `B`, `C`, and `D` are arrays. In this case, the output of a floating-point unit can be fed directly to the input of another floating-point unit to form a longer pipeline. This allows two floating-point operations to be executed at the same time, and a peak performance of 12 Mflops for 64-bit floating-point operations can be achieved. There are also scatter and gather operations to handle array operations with random array structures.

There are several special 32-bit registers to support vector operations: a vector length register, a vector increment register, and a mask register which holds the vector mask. There are also 8 32-element vector registers in each CE, each element being 64 bits wide. The result of a vector operation has to go to a vector register. There is a 16 Kbyte instruction cache in each CE which can provide a very high hit ratio to shorten instruction fetch time. An address translation lookaside buffer is also provided to facilitate address translation from a virtual address to its physical address.

Each CE may be used individually or as part of a complex to execute application programs in parallel. A complex consists of two or more CEs and the concurrency control bus, and is designed to execute FORTRAN-like do loops in parallel. This execution proceeds as follows. At the beginning of the parallel loop, processor `i` is assigned iteration number `i` of the do loop. As soon as a processor completes executing an iteration, it requests the next unassigned iteration number from the concurrency control bus. This procedure, called self-scheduling, continues until all the loop iterations have been assigned. At this point, the processors completing iterations are placed in the idle state, except for the processor executing the last iteration of the loop which, upon completion of all other iterations,
proceeds to execute the instruction immediately after the parallel loop. Notice that the different iterations of a parallel loop, when executed by a complex, are scheduled non-preemptively by the hardware. Iterations that have not been assigned do not begin executing until one of the assigned iterations completes. This approach to parallelism has been implemented via software in other machines such as the Cray X-MP [BoM88], Sequent [Ost88], and RP3 [SDNP85], and is known by the name of microtasking.

An important characteristic of microtasking is that a parallel construct can execute independently of the number of processors available. Two types of loop synchronization schemes are very useful. One is critical section locking, and another is synchronization from lower number of processors available. Two types of loop synchronization have been implemented via software in other machines such as the Cray X-MP [BoM88], Sequent [Ost88], and RP3 [SDNP85], and is known by the name of microtasking.

Notice that synchronization from higher to lower iterations will lead to deadlock when, for example, there is only one processor available to execute the parallel loop [TYF87]. The Alliant FX/8 concurrency control bus provides hardware assistance for cascade synchronization. It is possible to have a complex consisting of n processors and s - n detached processors in a cluster. The complex as a whole and each detached processor can be independently allocated for executing different programs.

For a multiply-nested loop, the Cedar system provides a higher level self-scheduling facility which allows each cluster to self-schedule an outer loop among clusters. Within each cluster, microtasking can then be applied to the inner loop nests. This capability is supported in the Cedar Fortran language through a loop structure called SDOALL. In an SDOALL loop, tasks will be created for each cluster to execute iterations of the loop, with intercluster communication via shared global memory.

2.2. Memory Hierarchy

The memory hierarchy in the Cedar system includes an I/O system, a shared global memory, a cluster memory and a shared cache in each cluster. It supports a virtual memory system with a page size of 4 Kbytes. The address translation is performed in each CE to translate a 32-bit virtual address to a 32-bit physical address. This allows it to address up to 4 Gbytes of physical memory. The physical address space is divided into two equal halves: 2 Gbytes for the memory within the cluster and 2 Gbytes for the global memory. The most significant bit is used to specify if a memory address is in the global memory address space or in a cluster memory address space. Notice that the 2 Gbytes of the cluster memory address space is visible only within each cluster. One cluster does not have access to the cluster memory address space of another cluster.

Between the cluster memory and CEs, there is a 128 Kbyte, 4-way interleaved shared write-back cache. All 8 CEs in a cluster share four ports to the shared cache. However, the bandwidth of the shared cache matches the issuing rate of the CEs unless severe conflicts occur. Each cluster has up to 32 Mbytes of cluster memory in the current prototype machine.

The shared global memory has multiple memory modules. Array data are stored across all memory modules to reduce memory conflicts and to allow data to be accessed in parallel to enhance global memory bandwidth. There is a SECDED (Single Error Correction Double Error Detection) code on each 64-bit memory word. Long memory access delay to the global memory is unavoidable in such systems. There are several reasons for this:

1. Due to physical pin limitations in packaging, it is extremely expensive and difficult to have a very wide data path in the global interconnection network. This prohibits large amounts of data from being transferred in parallel between processors and the global memory.

2. In a multiprocessor system, processors are accessing global memory independently. Memory conflicts are unavoidable regardless of the array storage schemes.

3. There will be network conflicts as requests are being routed through the interconnection network.

4. Due to cost effectiveness, larger and slower memory chips in the large shared global memory are usually used.

This problem has to be dealt with effectively to keep global memory from becoming a bottleneck. In Cedar, the long global memory access delay can be offset in two ways:

1. For programs with good locality, the cluster memory with a smaller and faster shared data cache can be used to minimize the number of global memory accesses from a CE. The instruction cache in each CE will also provide fast instruction fetches for each CE. Using compiler techniques, programs can be restructured to enhance program locality for data, e.g. by blocking and merging loops [Abus81]. This improvement of program locality will reduce the traffic in the network, which in turn will reduce the network and memory conflicts and improve global memory access delay.

2. Array data can be prefetched into local buffers in the global interface modules before they are needed, so long memory access delay can be overlapped with other useful operations. In Cedar, specific data prefetching instructions are provided to allow data prefetching to be done as early as possible. Furthermore, the global interconnection network allows requests to be pipelined through each stage of the network. So after a startup time, data can stream through the network at a rate that closely matches that of a CE.

Fast local memory (e.g. cache) has been used in high performance uniprocessor systems for a long time. In a
parallel system, multiple copies of the same data in different caches will create coherence problems if the data are updated at different instants of time in those caches. Hardware can be built to maintain coherence by automatically updating all of the copies in different caches with appropriate sequence control with respect to local reads and writes. In the Alliant cluster, those caches for floating-point processors and for I/O processors are managed by hardware to maintain coherence using the cluster memory bus. However, it is extremely costly and slow to maintain coherence among all clusters when a large multi-stage interconnection network is used in the system.

Notice that read-only data and the private data that are not shared among processors can be brought into local stores without causing data coherence problems. This will include program code and those variables that store temporary results during program execution. Only those shared data that will be updated at one time or another by processors in different clusters will have data coherence problems.

Furthermore, in a multiprocessor system, when processors are assigned to work on the same program, e.g. loop iterations scheduled on many processors, it is rare that a shared variable will be fetched and stored in a random order among processors (except for some functions in the operating system). Synchronization is usually required to maintain a correct read/write sequence among those processors sharing the same variable. A processor has to wait if the shared data it needs has not yet been produced (or stored) by another processor.

In Cedar, part of the memory hierarchy is visible to users through the Cedar Fortran language. Each variable in a program has memory attributes associated with it. The data which may cause coherence problems between clusters should be declared as global variables and would then only be stored in the global memory. The compiler can be used to identify variables that may cause coherence problems through complete data dependence analysis. This will be described in more detail in section 4.

The global memory modules in Cedar include a *synchronization processor* (SP) whose function is to make possible the execution of atomic operations on pairs of memory cells. In Cedar, two consecutive memory words can be considered as a single entity. One cell is used to store data and the other to store a synchronization key. The SPs implement critical sections that guarantee mutual exclusion in the access to each such entity. These critical sections, whose execution is requested by the CEs via packets, operate on the key cell (typically increment and decrement) and may also fetch or modify the data cell or the key cell. A boolean expression, tested inside the critical section, controls whether the other statements in the critical section are executed or not. This boolean expression is restricted to a comparison of the value of the key cell with a constant. This synchronization mechanism has several applications, some of which are discussed in detail in [ZhiYe87]. An example is presented below in Section 4.

### 2.3. Global Network (GN)

The Cedar global network consists of two unidirectional multiple-stage shuffle-exchange networks. One goes from the processors to the memories and the other in the opposite direction. The network is a packet-switching network. Each packet can have up to four 64-bit words with the first word always an address/control word. The network consists of 8 X 8 crossbar switches, organized into two stages that are connected using the perfect shuffle scheme, with hardware queues at their entry ports. Each port is 80 bits wide: 64 bits of data, 8 parity bits and 4 duplicated control signals which allow handshaking between network stages. The 4 control signals are duplicated to enhance the reliability of the network.

This type of interconnection was chosen because it is a good compromise between a bus scheme, with its performance limitations, and a full crossbar switch, whose cost can be prohibitive since it grows proportional to the product of the number of processors and memory modules. The Cedar network can provide a bandwidth close to that of a crossbar switch at a much lower cost [CLPY81]. The network has a cycle time of 85ns which provides a total bandwidth of 2.9 Gbytes/second for a 4-cluster 32-processor Cedar system.

The routing scheme in the Cedar interconnection network is a tag control scheme proposed in [Lawr75]. According to this routing scheme, in a 4-cluster Cedar system with 32 CEs and 32 global memory modules, each input-output pair of the network can have two possible paths between them. We can use both paths to improve network bandwidth and reliability, i.e. when one path is blocked due to a network conflict or a failure, the other path can be used. However, in the current prototype machine, only one path is used at any time. This is to avoid a “racing” condition that can occur in a multiple-path network, i.e. two requests issued from the same CE to the same memory location may reach the memory location out of order through two different paths. This can cause a serious problem. For example, assume the first request is a write operation and the second request is a read operation, if the order of these two memory operations is non-deterministic, the final outcome of these memory operations is also unpredictable. One way to avoid that is to force a CE to access the global memory sequentially. It will allow a CE to issue a memory request only after it receives an acknowledgement from the global memory indicating that the previous memory request has been completed. The performance can suffer quite significantly because the latency of a global memory access is an order of magnitude longer than a memory request to the shared cache. A
unique network path between a CE and each global memory module guarantees that the memory requests from a CE will be serviced in the order issued from the CE, so several requests can be outstanding at the same time to allow global memory access time to be overlapped. Memory synchronization is needed only when different CEs are trying to access the same memory location. This subject will be discussed in more detail later.

2.4. The Global Interface

One of the major modifications done to the Alliant FX/8 to convert it into a Cedar cluster was to expand the switch between the CE's and the 4-way shared CE cache from an 8 x 4 crossbar switch to an 8 x 8 crossbar switch. The four additional ports are used to connect each CE to a global interface (GI). Two GIs share one port of the crossbar switch. Each GI serves one CE and handles several important functions.

The most important function is to provide a pathway between a CE and the global interconnection network. A processor has to complete an address translation and produce a physical address before a request can leave the processor. The most significant bit of a physical address is then used to determine if the request should be routed to the shared cache or to the GI for a global memory access. The GI has to interact with the shared cache and the CE to control the crossbar switch.

There is a prefetching unit in each GI which handles the prefetching of array data from the global memory. The prefetching unit includes a 4 Kbyte data buffer arranged in 512 64-bit words, and a set of special registers similar to the ones in a CE that can specify the length, the stride and the mask of the array to be prefetched. A special prefetch instruction will start the prefetching and the returning data will be stored in the data buffer. Notice that the array data may return out of order due to possible memory and network conflicts. Hence, each word in the data buffer has a full/empty bit associated with it. When the prefetching first gets started, the data buffer is flushed and all of the full/empty bits are reset to empty. A returned word will set the corresponding bit to full. Using this scheme, the CE need not wait until all of the words are fetched before it can begin using them. A CE can start accessing the data buffer right after the prefetching is started. If a word is not back yet, i.e. the full/empty bit is still empty, the request will be pending in the prefetching unit until that word is fetched from the global memory.

The global interface also contains a synchronization unit which acts as a staging area for issuing synchronization instructions. A counter is used to record the number of pending write requests to the global memory. The counter is incremented each time a write request is sent from the GI to the global memory. A global memory module will send back a write acknowledgement to the GI after the write operation is completed, and the counter in the GI is then decremented. A synchronization instruction can be issued only after all of the pending requests to the global memory are completed. Since a read operation will always get a datum back from the global memory, the completion of a read operation is implicit. However, a CE normally can continue without waiting for the completion of write operations to improve its efficiency. However, a critical region enforced by a synchronization instruction must ensure that all of the read and write operations are completed before a CE enters or leaves the critical region. The write counter in the GI makes sure that all of the pending write requests are completed before a synchronization instruction is issued. Also, a synchronization instruction always gets some response from the global memory before the CE can continue.

3. The Xylem Operating System

The Cedar system provides three levels of parallelism: vector instructions, intracluster parallelism, and intercluster parallelism. The first two are supported by the hardware and are directly accessible by the programmer. This can be done using assembly language or by explicit constructs in Cedar Fortran. In addition, the compiler may implicitly generate appropriate or optimized code to access these hardware features. Intercluster parallelism is supported in the software of the Xylem operating system kernel [Emra85] and Cedar Fortran run-time library. This section will focus on the design and development of Xylem and overview the key features supporting intercluster parallelism and the management of the Cedar memory hierarchy.

Xylem is based on Alliant's operating system, Concentrix, which is in turn based on Berkeley's 4.2 implementation of UNIX. This evolutionary development path was chosen for many reasons. The Cedar system is intended to be general purpose in nature, and a multi-programming operating system is pretty much a foregone conclusion in order to effectively and economically utilize the computational capacity of the system and to satisfy the computing requirements of the software and applications development staffs of the project. Unix has become a popular operating system and many of the staff members were already familiar (or expert) with 4.2 Unix. Most importantly, Alliant's system provides considerable support for the devices, processors, and memory subsystems of the Alliant machine.

3.1. Design Goals and Development

The primary goal of the design of the operating system was to provide the ability for a single application, ideally written in the form of a single program, to effectively utilize the entire Cedar system at one time. Part of this goal involved providing the program with access to chunks of cluster memory in each cluster as well as shared access to global memory (at least that part
allocated to the program). Meanwhile, we had to maintain something that at least resembled a Unix environment so that it could support much of the software base available for Unix.

In summer 1985, the project acquired two Alliant machines, which would eventually become clusters of the Cedar machine. One of these machines had a sizable configuration and was put to immediate use doing performance experiments and applications software development. It quickly became evident to the operating system developers that compatibility with the Alliant system had to be maintained. This would be necessary to provide a smooth transition for (approximately 100) users as the Alliant machines are absorbed into the Cedar configuration.

As a result of those early experiences, the design was strongly influenced in the direction of extending Alliant's system to meet the needs of Cedar. We realized that the need and the feasibility of using the bulk of the Alliant kernel along with its scheduler. The Alliant system controls things internally on each cluster, while being directed by external events (i.e., conditions in global memory or cross-cluster interrupts). It can also be prioritized to favor Xylem (Cedar) programs. At the same time, each cluster will appear to be an independent "bare" Alliant system, supporting the entire mix of Concentrix processes, such as shells, mailers, and networking utilities, in addition to servicing Xylem programs.

This approach is proving to be effective in two ways. First, since the Xylem kernel will require a debugging period after the machine has been assembled, some of this debugging can be carried on while the clusters are independently available with the same system users are familiar with. Second, we have developed a Xylem kernel which runs on a bare Alliant and simulates a one- cluster Cedar by treating part of the Alliant memory as if it were Cedar global memory. This development kernel has been in steady use on one of the Alliants since summer 1986.

In the meantime, we have acquired two more Alliants, and for some time we had three (ethernetted) machines in use by software developers, all destined to "disappear" into the Cedar system. All these machines have been running our "one-cluster" Xylem system since October 1987, and provide most of the functionality that Xylem will deliver in a multi-cluster Cedar. Unfortunately, the problems of achieving extraordinary performance as well as maintaining software compatibility are somewhat at odds with each other.

3.2. Multi-tasking and Scheduling

The most important feature in Xylem is the abstraction of a process consisting of a number of parallel cluster tasks. Our notion of process is much like that of Unix's, essentially representing the entire state of execution of a program. However, the process itself doesn't execute, rather, individual cluster tasks execute, potentially on different physical clusters. Any cluster task of a Xylem process can create new cluster tasks, which become siblings of the creator, all belonging to the same process. There is no hierarchy between the tasks of a process. They are identified by small integers, unique only within the process.

Each cluster runs a copy of the same Xylem kernel, communicating primarily via data structures stored in global memory. There is a private copy of all the Alliant kernel data structures on each cluster, so each cluster can be scheduling Concentrix processes independently. A Xylem process is created when the exec system call is invoked on a Xylem format binary file. When this is done, the data structures for a single cluster task are set up in global memory.

There is a Xylem server process that has one task bound to each cluster. One of the server tasks will see a new cluster task first and may assign it to its cluster, based on global knowledge. From the Xylem process and task data stored in global memory, the server builds what looks very much like a Concentrix process and installs it in the Alliant data structures in the cluster. At this point, the cluster task is scheduled by the Alliant scheduler along with existing Concentrix processes. Remember that intracluster parallelism is provided by the Alliant hardware, and that the Alliant system already provides the support needed to manage this. The Xylem servers can also de-assign cluster tasks, causing them to be removed from their current cluster and assigned to another cluster, thereby achieving load balancing.

It can be noted that our abstract view of the execution of a program is very similar to the abstractions provided by the Mach operating system [ABBG86, TRGB87]. The Xylem process is much like the Mach task, essentially representing the system resources allocated to (the execution of) a program. What we call a cluster task (or just task) is similar to a Mach thread and represents an execution state of the underlying hardware. However, there are differences between these abstractions. For example, a Xylem task can utilize all the processors in the cluster, so there can actually be 8 different processor (CE) states, each of which also resembles a Mach thread. While the implementation of Xylem and Mach are very different, and of independent design, the striking similarities of the abstractions are intriguing. Whether these stem from similar educational and experiential backgrounds, or are a result of similar design goals leading to similar solutions, is difficult to say.

3.3. Memory Management

Another important aspect of Xylem is the virtual memory management it provides. Each page in the virtual memory system has access, location, and sharability attributes. The access attribute simply specifies whether the page is readable, writable and/or executable. Naturally, pages that are not writable are shared by multiple
tasks and processes.

The location of a page indicates which level of the physical memory hierarchy the page should reside in when actually referenced by a processor. This can be global or cluster. If a variable is declared global, it is placed in a section that is made global. The pages of that section will be loaded into global memory when the program executes. Likewise, cluster variables end up being loaded in cluster memory when they are referenced.

The global memory can be and is used as a backing store for cluster memory, while disk is used as backing store for global memory. When a page fault occurs on a cluster page that has been backed out to global memory, it is moved into cluster memory before restarting the faulting instruction, rather than just pointing the memory any particular data item is actually accessed from.

The sharability of a page specifies whether the page is shared between all the cluster tasks of a process, or is private to just one cluster task. If a page is shared, then every task in the process will see the same physical page at the same virtual address. When a new cluster task is created in the process it will be given access to all the shared pages that all the other tasks in the process already have access to. A page that is private (and writable) is never shared by multiple tasks.

When a new task is started, it doesn't get access to any of the private pages of any of the tasks in the process. Rather, it gets access to fresh copies of initial private pages in the binary image. This is another area in which Xylem and Mach differ. In Mach, all the threads within a task completely share a single virtual address space and are unprotected from each other [TeRa87], whereas in Xylem, each cluster task in a process may have private space in addition to space that is shared across the process. For example, the processor stacks are kept in private memory. As a result, we tend to think of a cluster task as being a "middle-weight" process, not having all the burden of firing up an entire process, but not quite as light as a Mach thread. In Cedar, the intracluster parallelism provided in hardware gives rise to fly-weight processes.

The notion of partial sharing, that is, having some pages that are shared between a proper subset of the tasks in a process, while protected from the other tasks, was considered. Also, having the same physical memory (pages) accessible by different tasks using different virtual addresses in each task was considered. Both of these ideas have been seen elsewhere and are feasible on the Cedar hardware. However, we felt that little, if any, performance enhancement could be achieved with these features, at the expense of additional complexity, memory requirements, and implementation time. We feel that our simpler memory model will be sufficiently powerful and easier for programmers to work with, particularly when it comes to debugging.

In addition to being able to specify memory attributes statically, virtual memory with arbitrary attributes can be allocated dynamically, and the access and location attributes of any page may be changed at run-time. All these capabilities are accessible with the single system function memctl, which takes a specific request and address range as arguments. A program (or library routines) can create a shared heap anywhere in the virtual address space and any task can extend the area as necessary. The other tasks do not need to "attach" to the new area. Addresses can be passed between tasks since the same virtual address will always map to the same physical memory for shared pages. Similarly, each task can create its own private heap anywhere, as long as it doesn't conflict with any previously allocated space.

With these facilities, it is also possible to use cluster memory to cache shared pages, and maintain coherency by (unprivileged) software means. Such pages would have the cluster and shared attributes. The kernel will allow one task to access such a page, but the second task that tries to access the page while it is in the cluster memory is the first will get trapped. This trap can be caught and recovery can be to change the page from cluster to global, thereby making it a single shared copy. A detailed description of these mechanisms may be found in [McEm87] along with some examples.

4. The Cedar FORTRAN language

The Cedar FORTRAN language is an extension of Alliant FORTRAN. The constructs it introduces give the programmer access to the main Cedar architectural features. This Cedar specificity can be considered a drawback of the language since it hinders portability. However, it was felt that the benefits of having an unobtrusive language outweigh the drawbacks caused by the lack of portability.

In Cedar FORTRAN, variables and arrays may be declared with the attribute sync to specify that the variable or each element of the array are structures with two components: data and key. Only variables and elements of arrays declared sync may be used as resources in conditional critical regions (see below).

Variables and arrays may also be declared global or cluster. These declarations determine the location attribute of the page where the variables and arrays are allocated. In Cedar FORTRAN, the sharability attribute may not be explicitly declared. All variables and arrays are assumed to be sharable. A possible (addressing) space optimization is for the compiler to allocate variables and arrays that are not accessed by more than one cluster task in private pages. Interprocedural analysis would improve the quality of this optimization.

There are two types of common blocks in Cedar
FORTRAN: task common and process common. Each cluster task has its own copy of task common blocks, whereas there is only one copy of process common blocks. A similar classification of task common blocks is used by the Cray multitasking FORTRAN extensions [Cray85].

The request to spawn a new cluster task is made in Cedar FORTRAN by calling the ctkstart routine. The first parameter to this routine specifies the number of processors the programmer is requesting to be allocated for execution of the new cluster task. Depending on the form of this parameter, the number specified may be considered as a suggestion or as a requirement. If the request is a suggestion, the system may allocate fewer processors than specified in the parameter. The second parameter is the name of the routine where the task is to start execution. The remaining parameters to ctkstart are parameters to the new cluster task. These may be passed by value or by reference.

There are also constructs to express two types of loop parallelism. Both of these forms are suitable to be executed under microtasking. The first, doall [LuBa80], indicates that the iterations of the loop can be executed in parallel and that there is no restriction on the order in which these iterations may be scheduled for execution. This means that the only synchronization allowed in the body of a doall loop is that needed to control access to critical sections, since these do not impose any restrictions on the order of scheduling of the iterations. There are two doall constructs. One, cdoall, uses the processors assigned to the cluster task to execute its iterations, and, if the number of processors is greater than one, it uses the concurrency control hardware to start and schedule the loop. The other doall construct is sdoall and makes use of processor complexes to execute each iteration. A cdoall should be nested inside an sdoall to make use of the processors in the complex. An example of the use of the doall constructs is the following matrix multiplication program:

```fortran
sdoall i=1,n  
integer j
end sdoall  
cdoall j=1,n
C(i,j) = dotproduct(A(i,:),B(:,j))
end cdoall
end sdoall
```

The three levels of parallelism of Cedar are present in this example. The intercluster parallelism is expressed by the sdoall and the intraccluster parallelism by the cdoall. The declaration of integer j inside the sdoall specifies that there should be a different copy of j for each iteration of the loop.

The sdoall is implemented by the run-time library using the microtasking approach. When a cluster task reaches the beginning of a sdoall loop it may start or awaken other cluster tasks so that they may cooperate in the execution of the loop. Of course, the sdoall may also proceed sequentially if only the original cluster task is used to execute all the iterations. The number of cluster tasks used to execute sdoalls can be controlled by the programmer through service routines.

The other type of loop parallelism is doacross [Padu79] which allows cascade synchronization on top of the critical section synchronization allowed for doalls. Currently, Cedar FORTRAN only allows doacross to be executed inside a cluster (i.e. only cdoacross is available).

For synchronization, the language includes some subroutines implementing synchronization functions similar to those available in the Cray multitasking FORTRAN extensions. Also included is a restricted form of the conditional critical section construct [Hoar72] which has the form:

```fortran
with resource when test  
statement list  
end with
```

The resource, say r, is a variable or an element of an array defined as sync. In the body of the statement, accesses and modifications to %data and %key (the data and key components of r respectively) are performed in atomic form. The test denotes a boolean expression whose value must become true before the critical section may be entered. To illustrate the use of this construct, consider the sequential loop:

```fortran
do i=1,N
    A(K(i)) = C(i) + D(i)
end do
```

which can be transformed into the equivalent parallel loop:

```fortran
sync A
A(:,)%key = 0  
cdoall i=1,n  
with A(K(i)) when .true.
    if A(K(i))%key < i then
        A(K(i))%data = C(i) + D(i)
        A(K(i))%key = i
    end if  
end with  
end cdoall
```

Notice that since the vector K could have duplicated values, the parallel program has to guarantee that after the loop completes, each element contains the value assigned by the highest iteration. This is accomplished by keeping in the key the value of the last iteration that stored into the data part. Iteration i only assigns to the data part if the last iteration that modified the array element was less than i.

6. The Faust Programming Environment

Due to the additional difficulties presented by the
programming of parallel machines, the need for tools is even more urgent than it is for the sequential. These difficulties originate, in part, from the parallelism itself, and in part from other architectural features such as the possible presence of a user controllable memory hierarchy and the resulting coherency problems.

One important class of tools to help alleviate these problems is that of translators such as vectorizers and parallelizers [PaWo86]. These are not exclusively aimed at parallelizing existing sequential code, but are also useful tools when the program is written from scratch for parallel execution. In fact, a program that executes efficiently on a parallel machine is not necessarily an easy to understand, well structured program. With the help of parallelizers, programmers may concentrate on writing a readable program and leave at least some of the optimizations to the translator.

Translators may be classified into compilers and source-to-source restructurers. Most of the parallelizing translators provided by computer manufacturers are compilers. Restructurers have the advantage that it is easy for the programmer to see how the program was transformed, and to perform additional program transformations by hand (for example by changing the algorithm used in a part of the program) in those places where the restructurer did not do a satisfactory job. Even though most of today's restructurers are batch oriented, there is a growing awareness of the advantages of interactive program restructurers. These are usually coupled with performance monitors that detect which parts of the program consume the most time to help the programmer decide where to focus attention. Interactive restructurers conveniently allow the programmer to specify the type of transformation desired for a certain segment of code, and to carry on a dialogue with the restructurer.

Another important tool is the debugger. Besides the classical breakpoint debugger, other tools are needed to cope with the timing problems introduced by parallel programming. One such new tool uses dependence analysis and trace analysis to detect those segments of the parallel program where race conditions may occur. A race condition is present when, for example, on one path of the program a variable is fetched, while in another parallel path the same variable is modified, and the code does not guarantee any order of execution between these two accesses to the variable. A race condition is not necessarily an error since the program could be asynchronous. However, in many cases a race condition identifies a problem. A discussion of a debugging tool that detects races may be found in [AlPa87].

References


