Parallelization and Performance of Conjugate Gradient Algorithms on the Cedar hierarchical-memory Multiprocessor

Ulrike Meier and Rudolf Eigenmann *

Abstract

The conjugate gradient method is a powerful algorithm for solving well-structured sparse linear systems that arise from partial differential equations. The broad application range makes it an interesting object for investigating novel architectures and programming systems.

In this paper we analyze the computational structure of three different conjugate gradient schemes for solving elliptic partial differential equations. We describe its parallel implementation on the Cedar hierarchical memory multiprocessor from both angles, explicit manual parallelization and automatic compilation. We report performance measurements taken on Cedar, which allow us a number of conclusions on the Cedar architecture, the programming methodology for hierarchical computer structures, and the contrast of manual vs automatic parallelization.

1 Introduction

The preconditioned Conjugate Gradient Method is a powerful tool for solving sparse well structured symmetric positive definite linear systems that arise in many applications. This method has been considered on a variety of parallel computers, vector computers as well as parallel processors. In this paper, we investigate the efficiency of some suitable preconditioned Conjugate Gradient schemes implemented on Cedar, a parallel computer which possesses besides three levels of parallelism (clusters, processors and vectorization) the following interesting memory structure: a global memory shared by all clusters, and a cluster memory and a fast high-performance cache which are local to each cluster but shared by the cluster's processors. Former experiments on an Alliant FX/8 (which is equivalent to a cluster of Cedar) have shown that the performance of iterative methods on one cluster is limited by its cache size [MS88]. But the use of this new architecture showed an improvement in performance. Data locality could be increased significantly by distributing the data across clusters and handling smaller chunks on each cluster. We implemented the classical Conjugate Gradient scheme (which is the slowest), but also the simplest and therefore a good algorithm to gain a better understanding of the performance behavior of Cedar. We also examine a blockdiagonal block Incomplete Cholesky preconditioner as well as the reduced system approach.

Another aspect considered here is automatic parallelization of the conjugate gradient scheme. To generate an efficient object code the compiler needs to take into account the

*Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 305 Talbot Laboratory, 104 South Wright St., Urbana, IL 61801-2932. This research was supported by the National Science Foundation under Grant No. US NSF CCR-87-17942, and the Dept. of Energy under Grant No. DE-FG02-85ER25001, with additional support from NASA NCC 2-559 (DARPA).

available tools of parallelization as well as the complicated memory structure. The experiments with manual parallelization have shown that the choice of memory clearly affects the performance. A compilation method that takes advantage of the memory hierarchy is introduced and supported by experiments. Both approaches, the manual as well as the automatic one, are compared.

In Sections 2 the architecture and system software is briefly described. Section 3 describes the conjugate gradient schemes and Section 4 their implementation and experimental results. In Section 5 the automatic compilation is discussed. Section 6 compares manual and automatic parallelization, and finally section 7 contains the conclusions.

2 Cedar: a vector multiprocessor with a hierarchical memory

2.1 The Cedar architecture

Cedar consists of 4 multiple processor clusters which are connected through an interconnection network to a globally shared memory. To prevent long global memory access delays, array data can be prefetched into local buffers before they are needed. Each Cedar cluster is a slightly modified Alliant FX/8. Each processor has eight 64-bit 32-element vector registers. The processors are connected to a concurrency control bus and a cache which is shared by the processors in each cluster. The cache is connected to the cluster main memory through a memory bus with a maximum bandwidth of 188 megabytes/second. In the here considered Cedar configurations, each cluster has 32 mbytes of cluster memory, 512 kilobytes of cache memory and 4 processors. The cluster shared global memory is 32 megabytes. The final Cedar configuration is expected to have 8 processors per cluster and a global memory of 64 megabytes.

2.2 Cedar system software

The software that coordinates the clusters is the Xylem Operating System [Emr85], an extension of Unix. Its functionality is made available to the user through Cedar Fortran, the main application programming language. Cedar Fortran is basically Fortran77 with a few additional constructs for exploiting Cedar architectural features [GPHL90] . Users program Cedar by either writing directly in Cedar Fortran, or by starting from a sequential Fortran77 code and applying the auto-parallelizing Cedar Restructurer, outputing Cedar Fortran [EHJP90]. Important Cedar Fortran constructs, refered to in this paper are:

CTSKSTART forks a new task. The fork operation is costly, taking up to 200ms. It is used for initiating long-term parallel activities.

Synchronization primitives: A variety of constructs is available to manually perform synchronization. This can be done by defining events, integer variables in global memory, which can be activated by the routines evpost, evclear, evwait, or locking certain parts of global memory to prevent simultaneous writes by different clusters. For the manual parallelized code, we used the routine ifetch_and_add and some versions of evwait, evclear and evpost which make use of busy-wait methods. A more detailed description of the synchronization mechanism used is given in Section 4.1.

SDOALL, CDOALL are loop constructs that spread the iterations across all Cedar clusters, and across all processors of a cluster, respectively. Work parallelized in this manner executes in 'lightweight' mode, forking in 1 μs within and in 10 to 100 μs across clusters.

CLUSTER, GLOBAL type statements are for placing variables in the memory hierarchy. Cluster data can only be seen by the local processors. Data declared global are automatically prefetched in block mode when referencing them in vector operations.

Typically, application programmers think of a Cedar program as a number of explicit tasks, coordinated by synchronization calls, whereas the Cedar restructurer, translating a Fortran77 source code, transforms the sequential loop structure into as many parallel loops as possible. This is also true for the Conjugate Gradient algorithm. The explicit parallel implementation makes use of ctskstart() and synchronization constructs, while the auto-parallelized version applies SDOALL and CDOALL constructs. We will comment on some interesting differences of these two paradigms in section 6.

3 The CG algorithm

3.1 The CG family and its application range

The Preconditioned Conjugate Gradient method is a very effective method for solving sparse well structured symmetric positive definite linear systems which e.g. arise from finite difference or finite element discretizations of elliptic partial differential equations. Many efforts have been made to implement it with a variety of preconditioning techniques on different parallel computers, trying to take advantage of the various architectures [MS88] [vdV86]. As it consists mainly of vector operations, it turns out to be a very efficient method for a vector computer but due to the necessity of evaluating dotproducts in each iteration, it is not as well suited for a parallel computer that requires larger grain parallelism and has a distributed memory system. Among the preconditioners considered were many variants of Incomplete Cholesky factorization preconditioners which improve the condition number of the preconditioned system very efficiently [CGM85], are however highly recursive and not suited for parallel computation. Vectorization efforts improved the performance, worsened however the convergence of the method [Meu84]. Polynomial preconditioners [Saa85] were another attempt to combine higher convergence rates and a higher degree of parallelism, turned however out to be not as efficient as a high convergence rate requires a high degree polynomial which increases however the computational complexity. Another approach considered was the reduced system approach which leads to a system of half the size and is an efficient method if the matrix of the linear system is consistently ordered, i.e. the red and black points can be decoupled which is the case for a 5-point finite difference discretization. We focus here on the Classical Conjugate Gradient Algorithm which due to its simplicity is a good tool to evaluate the performance behavior of Cedar. We also consider a vectorized block Incomplete Cholesky preconditioner which is a very efficient and stable scheme if considered from the numerical point of view and the reduced system approach which is the best scheme among the considered.

3.2 The Classical Conjugate Gradient Algorithm

We consider a linear system of n^2 equations

$$\mathbf{A}\mathbf{x} = \mathbf{f}, \tag{1}$$

where $\mathbf{A}$ is a symmetric positive definite block tridiagonal matrix with tridiagonal $n \times n$-diagonal blocks and diagonal off-diagonal blocks.

This system can be solved iteratively by the conjugate gradient algorithm which is given here in its preconditioned form:

$\mathbf{M}$ is the preconditioning matrix which is symmetric positive definite, $\mathbf{x_0}$ is an arbitrary initial approximation to $\mathbf{x}$,

$$\mathbf{r} \leftarrow \mathbf{f} - \mathbf{A}\mathbf{x_0},\ \mathbf{p} \leftarrow \mathbf{r},\ solve\ \mathbf{M}\mathbf{z} = \mathbf{r},\ \gamma \leftarrow \mathbf{r}^{\mathbf{T}}\mathbf{z} \tag{2}$$

do until stopping criteria are fulfilled

$$\begin{aligned}
&\mathbf{q} \leftarrow \mathbf{A}\mathbf{p},\\
&\tau \leftarrow \mathbf{p}^{\mathbf{T}}\mathbf{q},\\
&\alpha \leftarrow \tfrac{\gamma}{\tau},\\
&\mathbf{x} \leftarrow x + \alpha\mathbf{p},\\
&\mathbf{r} \leftarrow \mathbf{r} - \alpha\mathbf{q},\\
&solve\ \mathbf{M}\mathbf{z} = \mathbf{r},\\
&\gamma_{new} \leftarrow \mathbf{r}^{\mathbf{T}}\mathbf{z},\\
&\beta \leftarrow \tfrac{\gamma_{new}}{\gamma},\\
&\mathbf{p} \leftarrow \mathbf{z} + \beta\mathbf{p},\\
&\gamma \leftarrow \gamma_{new}
\end{aligned}$$

end

For the classical Conjugate Gradient algorithm (CG), $\mathbf{M}$ is just the identity matrix, therefore solving $\mathbf{M}\mathbf{z} = \mathbf{r}$ is omitted and $\mathbf{z}$ replaced by $\mathbf{r}$ in the above algorithm description. The basic CG iteration can be vectorized very efficiently for well structured problems as the one considered here. The elementary operations required are matrix-vector multiplications, dotproducts and linear combination of vectors.

3.3 A block diagonal-block Incomplete Cholesky preconditioner

For this approach, $\mathbf{A}$ is approximated by a preconditioning matrix $\mathbf{M}$ which was obtained through the following construction: Partition A into a $k \times k$-block matrix where k is the number of clusters. Consider the block diagonal matrix obtained by taking the block diagonal of A (see Fig 1) and approximate each diagonal block by a block Incomplete Cholesky preconditioner. For the sake of vectorizability we chose the vectorized block Incomplete Cholesky preconditioner INVC3(1) [Meu84]. This preconditioner is completely parallelizable across clusters, has however the disadvantage that the number of iterations depends on the number of clusters used. This can also be seen clearly in our experiments (see Section 5).

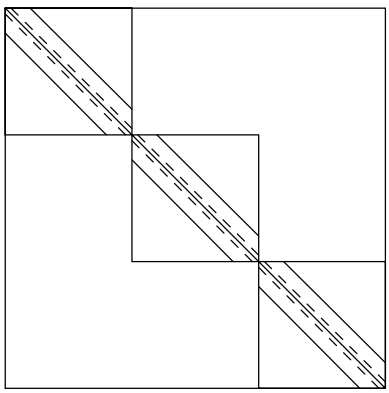


Figure 1: Block ICCG preconditioner

3.4 The reduced system approach

Another approach that was implemented on Cedar is the reduced system approach. Reordering the unknowns with the permutation matrix P according to a red-black coloring of the nodes we get the following linear system

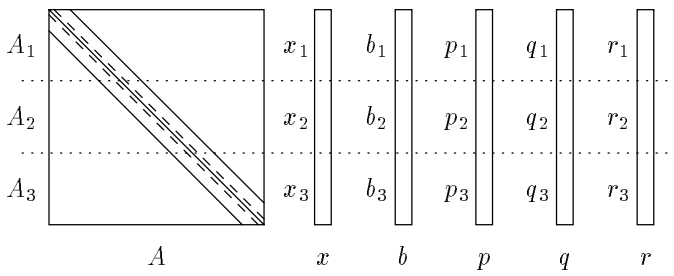
$$PAP^TPx = \begin{bmatrix} D_R & G \\ G^T & D_B \end{bmatrix} \begin{bmatrix} x_R \\ x_B \end{bmatrix} = \begin{bmatrix} f_R \\ f_B \end{bmatrix} = Pf$$

where D_R and D_B are diagonal and G is a well structured sparse matrix with 5 non-zero diagonals if n is even and 4 non-zero diagonals if n is odd. When scaled, the Schur complement of this system gives the reduced system $Cy = g$ of order $n^2/2$ for even n and of order $(n^2-1)/2$ for odd n with $C = I - H^TH$. The reduced system is now solved by the classical conjugate gradient. Once y is found, the solution x can easily be retrieved from y. The reduced matrix C has nine non-zero diagonals in the case of odd n. The main part of the algorithm, solving the reduced system, is implemented as described in the following section. The pre- and post- operations that are necessary to generate the reduced system and retrieve the solution of the original system are completely parallel and can be implemented with only few synchronization points.

4 Explicit parallel implementation on Cedar

4.1 Implementation structure

The algorithm was implemented for an arbitrary number k of clusters by splitting the matrix and the vectors as shown below (note that the $\mathbf{p_i}$ must be chosen slightly larger as shown here to compute $\mathbf{A_ip_i}$), and performing the operations involving $\mathbf{A_i}$, $\mathbf{x_i}$, $\mathbf{b_i}$, $\mathbf{p_i}$, $\mathbf{q_i}$ and $\mathbf{r_i}$ on cluster i, $i = 1, ..., k$. Not all of these operations are, however, independent from the



results obtained on the other clusters. For the computation of the dotproducts τ and γ_{new}, the partial results τ_i and γ_i have to be accumulated. For the evaluation of the matrix-vector product $\mathbf{A_ip_i}$ n elements of $\mathbf{p_{i+1}}$ and/or $\mathbf{p_{i-1}}$ which are calculated in cluster $i+1$ and/or $i-1$ are required. For the exchange of those elements, a work array w of length $2n(k-1)$ is created in global memory (GM). The original matrix, right hand side and the solution, as well as the variables γ_i, τ_i, $i = 1, ..., k$, the synchronization variables and w are in global memory. $\mathbf{A_i}$, $\mathbf{x_i}$, $\mathbf{b_i}$, $\mathbf{p_i}$, $\mathbf{q_i}$, $\mathbf{r_i}$, α, β, γ, γ_{new} and τ are created in cluster memory using dynamic allocation. The contents of $\mathbf{A}$ and $\mathbf{b}$ are copied to $\mathbf{A_i}$ and $\mathbf{b_i}$ in the beginning of the subprogram and the contents of $\mathbf{x_i}$ to $\mathbf{x}$ before returning to the calling program. The new iteration loop as implemented on k clusters (here $k = 3$) is of the form ($\star$ denotes

synchronization points):

Task 1	*Task 2*	*Task 3*
.	.	.
.	.	.
$\mathbf{q_1} \leftarrow \mathbf{A_1 p_1}$	$\mathbf{q_2} \leftarrow \mathbf{A_2 p_2}$	$\mathbf{q_3} \leftarrow \mathbf{A_3 p_3}$
$\tau_1 \leftarrow \mathbf{p_1}^{\mathbf{T}} \mathbf{q_1}$	$\tau_2 \leftarrow \mathbf{p_2}^{\mathbf{T}} \mathbf{q_2}$	$\tau_3 \leftarrow \mathbf{p_3}^{\mathbf{T}} \mathbf{q_3}$
$\star$	$\star$	$\star$
$\tau \leftarrow \tau_1 + \tau_2 + \tau_3$	$\tau \leftarrow \tau_1 + \tau_2 + \tau_3$	$\tau \leftarrow \tau_1 + \tau_2 + \tau_3$
$\alpha \leftarrow \frac{\gamma}{\tau}$	$\alpha \leftarrow \frac{\gamma}{\tau}$	$\alpha \leftarrow \frac{\gamma}{\tau}$
$\mathbf{x_1} \leftarrow \mathbf{x_1} + \alpha \mathbf{p_1}$	$\mathbf{x_2} \leftarrow \mathbf{x_2} + \alpha \mathbf{p_2}$	$\mathbf{x_3} \leftarrow \mathbf{x_3} + \alpha \mathbf{p_3}$
$\mathbf{r_1} \leftarrow \mathbf{r_1} - \alpha \mathbf{q_1}$	$\mathbf{r_2} \leftarrow \mathbf{r_2} - \alpha \mathbf{q_2}$	$\mathbf{r_3} \leftarrow \mathbf{r_3} - \alpha \mathbf{q_3}$
$\gamma_1 \leftarrow \mathbf{r_1}^{\mathbf{T}} \mathbf{r_1}$	$\gamma_2 \leftarrow \mathbf{r_2}^{\mathbf{T}} \mathbf{r_2}$	$\gamma_3 \leftarrow \mathbf{r_3}^{\mathbf{T}} \mathbf{r_3}$
$\star$	$\star$	$\star$
$\gamma_{new} \leftarrow \gamma_1 + \gamma_2 + \gamma_3$	$\gamma_{new} \leftarrow \gamma_1 + \gamma_2 + \gamma_3$	$\gamma_{new} \leftarrow \gamma_1 + \gamma_2 + \gamma_3$
$\beta \leftarrow \frac{\gamma_{new}}{\gamma}$	$\beta \leftarrow \frac{\gamma_{new}}{\gamma}$	$\beta \leftarrow \frac{\gamma_{new}}{\gamma}$
$\gamma \leftarrow \gamma_{new}$	$\gamma \leftarrow \gamma_{new}$	$\gamma \leftarrow \gamma_{new}$
$\mathbf{p_1} \leftarrow \mathbf{r_1} + \beta \mathbf{p_1}$	$\mathbf{p_2} \leftarrow \mathbf{r_2} + \beta \mathbf{p_2}$	$\mathbf{p_3} \leftarrow \mathbf{r_3} + \beta \mathbf{p_3}$
$\star$	$\star$	$\star$
exchange part of $\mathbf{p_1}$	*exchange part of* $\mathbf{p_2}$	*exchange part of* $\mathbf{p_3}$
.	.	.
.	.	.

The synchronization for the dotproducts was implemented by using the routine ifetch_and_add (which adds an integer to a global integer) and a wait-routine. Every cluster after having finished their part of the dotproduct adds an integer to the global integer to signal that they are finished writing and when all parts of the dotproduct are computed and the global integer equals the desired value, all tasks can read those values and accumulate the final dotproduct. A second global integer is necessary to indicate when all tasks are done reading to prevent overwriting of the partial dotproducts before they have been read by all tasks. For the exchange of the overlapping elements of p, $2(k-1)$ event variables and the routines evwait, evclear and evpost were used.

4.2 Numerical Results

Several versions of the described algorithms were implemented on Cedar.

The codes run are given in the following table.

code	description
CGM	Cg, k clusters, all arrays in CM
CGGM	k-cluster code, matrix in GM, other arrays in CM
RSCG	k clusters, all arrays in CM
BDBICG	k clusters, all arrays in CM

The following experiments were performed for the Laplace equation with Dirichlet boundary conditions on a $n \times n$-grid. The timings given in the appendix are wall clock times (in seconds) and the best out of at least 3 runs for each system size being the only user of the machine.

4.2.1 Convergence behavior of CG schemes

The reduced system approach is clearly faster than the classical conjugate gradient due to its smaller operation count. Only about half as many iterations are necessary for convergence

to the same accuracy (see Table 1), and the number of operations per iteration is reduced as the system is half the size of the original system. The number of iterations needed by BDBICG is clearly lower than those above. As expected, they increase if we use more than 1 cluster. For large system sizes however, the number of iterations are only slightly varying for more than one cluster. The speedups in Figure 2 show that the multicluster versions are still clearly faster than the one-cluster version.

	CG(G)M	BDBICG				RSCG
n		1cl	2cl	3cl	4cl	
31	91	16	24	29	31	45
63	180	32	42	47	50	89
95	269	46	57	64	68	133
127	358	61	74	80	85	177
159	446	75	90	95	102	221
191	534	88	107	113	118	265
223	623	101	123	131	135	308
255	711	114	140	148	151	352
287	799	126	157	165	164	395
319	887	139	173	182	179	438
351	975	152	190	198	194	481

Table 1: Number of iterations

4.2.2 Performance behavior

The performance (in Mflops) of RSCG and BDBICG is in general worse than the performance of CGM due to a larger overhead, caused by the less efficient generation of the reduced system or the preconditioning step and in the case of RSCG additionally the larger synchronization overhead. For large system sizes, the performance of RSCG and BDBICG is approximately 5*k Mflops where k is the number of clusters. For small system sizes, the performance is clearly lower and increases with increasing system sizes for BDBICG (which requires less synchronization due to its faster convergence) faster than for RSCG.

4.2.3 Influence of memory use on performance

The number of data elements in cluster memory per iteration is $7n^2$ data elements for CGM and $4n^2$ data elements for CGGM. In the following table the maximal n is given for which these data elements theoretically still fit in cache where CS is the cache size (here: $CS = m * 64K$ for m clusters), i.e. $n_{max} = \sqrt{CS/7}$ or $n_{max} = \sqrt{CS/4}$ respectively.

Theoretical and observed performance peaks				
number of clusters	theoretical for $7n^2$	observed for $7n^2$	theoretical for $4n^2$	observed for $4n^2$
1	96	63	128	95
2	136	127	181	159
3	167	159	221	191
4	193	191	256	223

The observed peaks for CGM ($7n^2$ data elements per iteration) and CGGM ($4n^2$ data elements per iteration) are slightly below the theoretical peaks which is due to the fact that the cache is not optimal (see figures) and the experimental increment for n is 32.

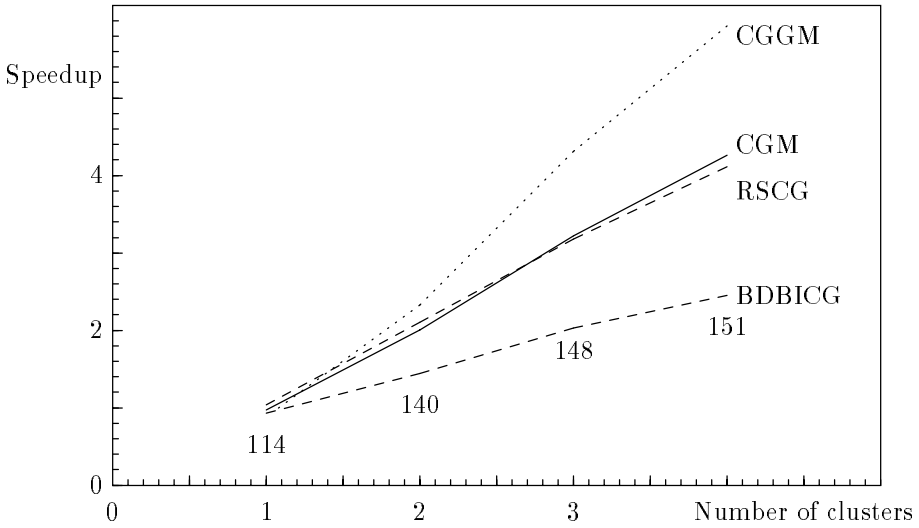


Figure 2: Speedups for Preconditioned CG

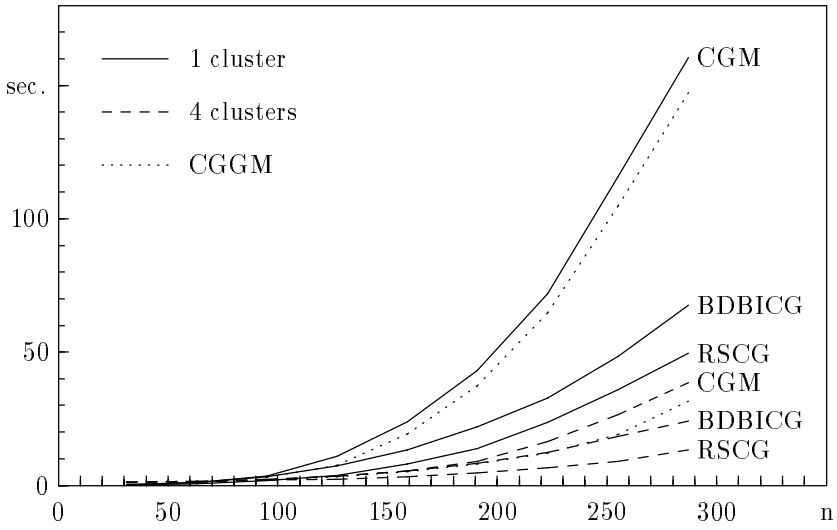


Figure 3: Times for Preconditioned CG on 1 and 4 clusters

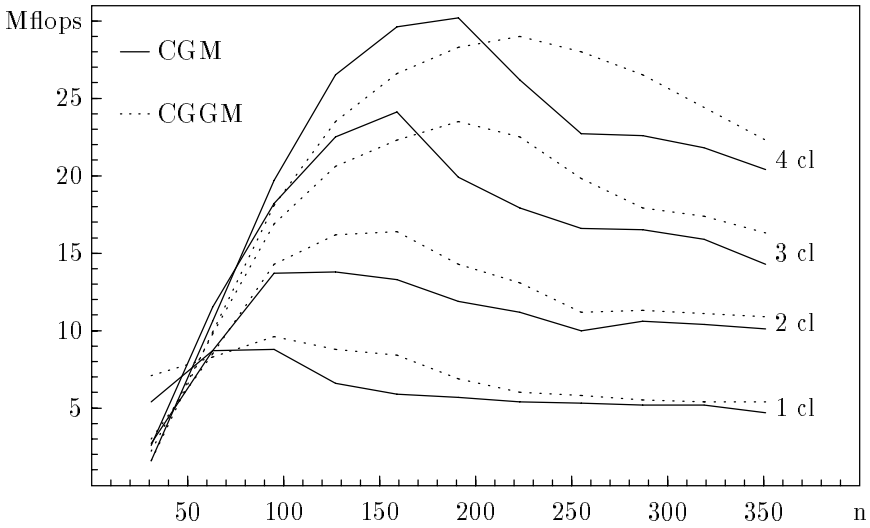


Figure 4: Mflop rates for CGM and CGGM on Cedar

5 Automatic compilation for Cedar

The work reported in this section is part of the Cedar Fortran Compiler project [EHJP90]. The auto-parallelizer component is built upon the Kuck & Associates, Inc. proprietary Fortran restructurer Kap [Kuc88][HMD+86]. In our first project phase we were mainly concerned with applying traditional compiler techniques and measuring their effectiveness on Cedar. This corresponds roughly to the scheme described in section 5.2.1. Currently our main interest is in developing data distribution schemes for better exploiting Cedar's cluster memories and in refining the compilation techniques using a broad spectrum of test applications. The Conjugate Gradient algorithm is an important part of our benchmark suite. It showed us upper limits of the achievable performance and gave much insight in the Cedar system's intricacies.

5.1 Recognition of parallelism for automatic compilation

The CG loop structure consists of a sequence of small kernels whose independent iterations can be recognized by simple data dependence tests. The only non-trivial parallelizable constructs to recognize are the dotproduct kernels. Their pattern, and the appropriate transformation, are well known, though. Hence, the recognition of parallelism in the CG is a straightforward process. Similar is true for the automated generation of a parallel implementation, provided we apply 'traditional' techniques. Each kernel can be turned into a parallel loop, spreading the iterations across processors.

A much less understood area in parallelizing compilation is entered once we attempt to partition data and distribute them to different processor or processor clusters. There are only few known approaches, most of which include user assistance for the partitioning process [], or tackle the problem at a theoretical end, not yet proven useful in practice [GJG88]. Our approach here shall be to find heuristics that deal with significant program patterns.

The regular computational structure of the CG allows us to divide the loop index spaces into 4 chunks and then copying to cluster memory read-only data and data that will be read from the same partition it is written to. In the main part of the CG algorithm this access pattern applies for all but one arrays.

A more advanced technique would recognize the data volume that needs to be exchanged at runtime between clusters, thus, distributing the remaining array. Still, this recognition can be easy for the CG, since the volumes are simple-to-describe regions of linear data spaces.

Distributing the dotproduct kernels is slightly more complex. A dotproduct can be transformed so it accumulates partial sums within each cluster, followed by a synchronized global sum of these parts.

5.2 Schemes for automated parallel implementation on Cedar

5.2.1 A shared-memory oriented transformation

We have described the process of recognizing and implementing parallelism in CG, using traditional techniques, a simple one. The Cedar restructurer transforms all kernels of Figure 5 into Cedar Fortran as in the following 'daxpy' example.

```
double precision a(n), b(n), c          double precision a(n), b(n), c
do i=1,n                                global a,b,c
   a(i) = a(i) + c* b(i)        -->     sdoall i=1,n,256
enddo                                      integer j
                                           cdoall j=i,i+255,32
                                             a(j:$32) = a(j:$32) + c * b(j:$32)
                                           end cdoall
                                        end sdoall
```

The compiler has decided to place the arrays a,b and the scalar c in global memory, so they can be seen by all processors of the Cedar complex. The loop is first spread across all clusters, in chunks of 256 iterations. The clusters, in turn, spread the work across all 8 processors, which execute the daxpy statement in a vector operation of length 32.

There are a number of issues involved in this transformation process, such as choosing the right chunk-size, making simpler transformations if 'n' is small, not making an SDOALL if placing data globally causes performance loss elsewhere, etc. Also, in practice, this code looks less readable. For example, it contains additional computation to deal with the case that 'n' is not a multiple of 256. A more detailed view of the compiler is given in [EHJP90].

5.2.2 Applying data partitioning and distribution techniques

Using the above scheme, Cedar is not yet fully exploited. There are only few variables placed in cluster memory, namely variables used but inside one sdoall iteration, in which case they are declared sdoall-local. In our example this holds for the cdoall loop index 'j'.

In order to take fully advantage of the local memories, the main data structures should be placed there. In section 5.1 we have introduced simple heuristics allowing the partitioning and data distribution in the CG algorithm. All arrays, except 'p' were divided by the number of clusters and the partitions are copied from global to cluster memory before the CG main loop, as illustrated in Figure 6. Read-only data can be localized evidently. The compiler's main task is to identify program phases that contain read-only access patterns. For the CG main loop, 30% data references are read-only. Read-write data can also be localized if reads go to the same data partition as writes, i.e. there is a 'simple index function' such that no cross-cluster dependencies arise. This holds for 20% of the data references. The compiler's task is also to judge whether copy operations, for moving the data from global to cluster memory and back, can be paid off by the number of local accesses within the candidate program phases.

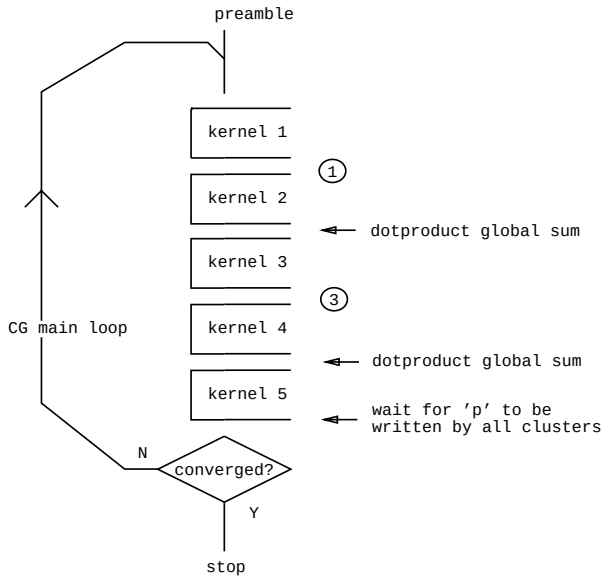


Figure 5: The structure of the CG computation

We are currently implementing this scheme in the compiler. The performance data presented in Figure 7 result from a manually applied version of the techniques.

5.3 Performance results

Figure 7 shows speedups we measured for the shared-memory oriented (solid lines) and the data-partitioning compilation scheme (dashed lines). Data sizes are 255 by 255 and 63 by 63, respectively. Speedup one is measured as the program performance on one cluster using cluster memory. Thus, 'speedup' can be looked at as a measure for what we gain when going from a traditional shared-memory architecture of type Alliant FX/4 *, to a Cedar type machine.

We show two sets of measurements which actually correspond to two different Cedar configurations. Both configurations have 16 processors. However, the global memory size is 16 MB in configuration 1 and 32 MB in configuration 2. The raw global memory bandwidth is proportional to its size.

The measurements represent best timings of the CG-main-loop iterations in Figure 5. Our measurements showed that only about 10% of the iterations are significantly slower than the best timings. This rate can go up if computing resources are used by OS server processes or if they are shared with other programs.

We can characterize Figure 7, configuration 1 as follows:

The 'global-memory strategy' works well until the memory bandwidth is reached. On two clusters we get twice the speed of one cluster. On three and four clusters the global memory becomes the bottleneck. The resulting best speedup is about 2.3.

The data-partitioning scheme achieves near-ideal speedup up to 4 clusters This scheme successfully avoids the memory saturation. We observe a near-linear behavior, close to the one for the manually optimized program.

*the final Cedar will have 8 processors per cluster, thus, compare to an FX/8

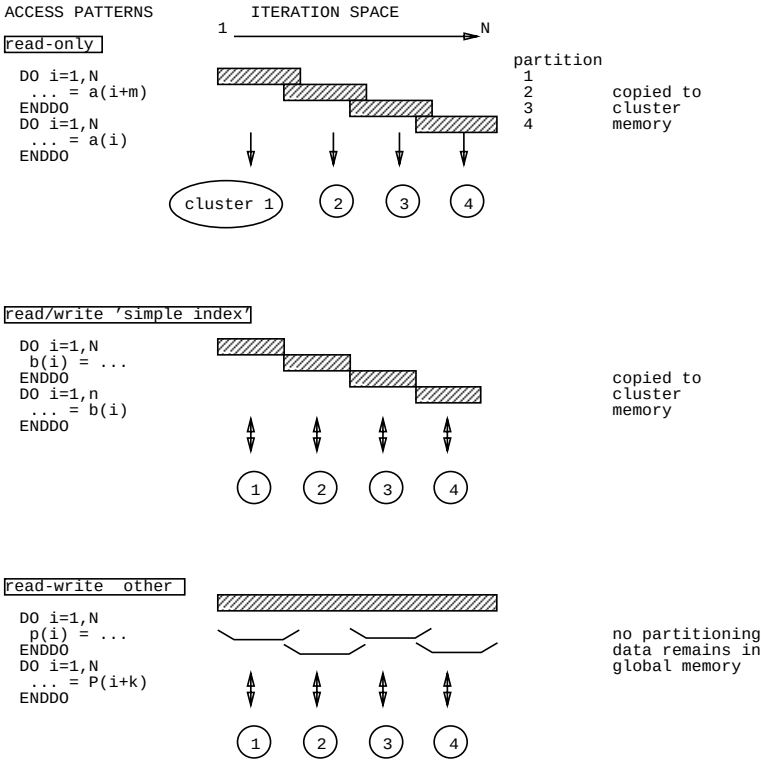


Figure 6: Data partitioning and localization

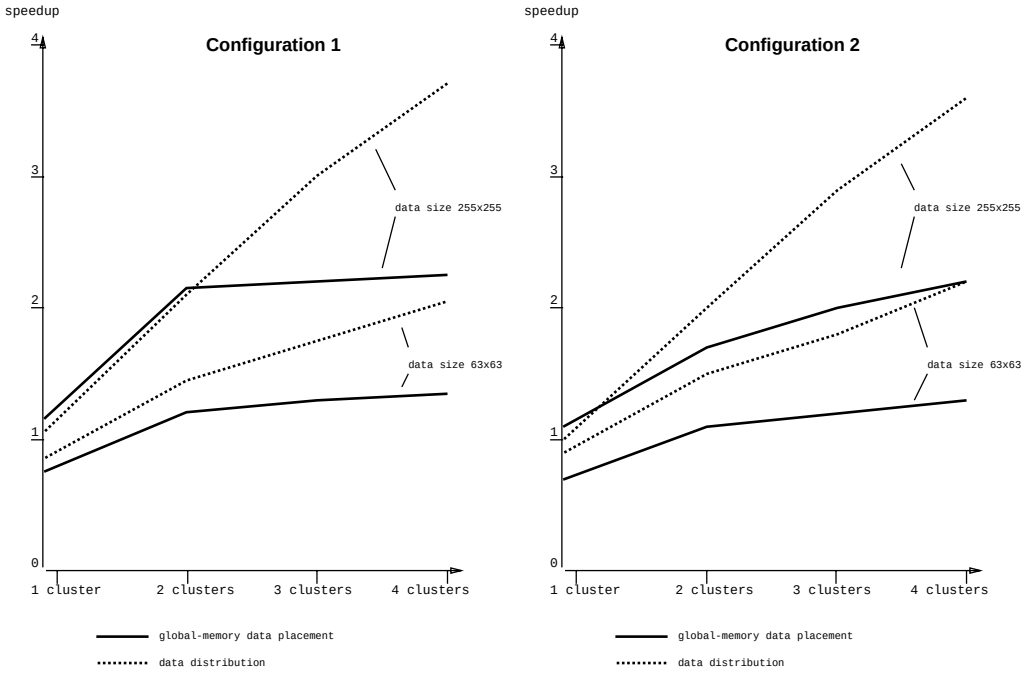


Figure 7: Compiler performance characteristics

Note that still 50% data references go to global memory. This is apparently low enough to keep the memory contention at a low level. In addition, we benefit from the fact that accesses to global and local data can overlap, because Cedar has separate data paths to shared and cluster memory.

Small data sets cause a significant deterioration There are two main effects responsible. First, the kernel execution times go down to a few milliseconds, where the loop control overhead becomes a factor. Second, all data fits in one cluster cache, making the one-cluster/cluster-memory execution relatively fast. Note that the 1-cluster 'speedup' in Figure 7 is at 0.8, demonstrating the speed difference of global vs cache memory. The first effect goes along with the number of synchronization points required. By fusing the kernels, as described above, the effect was reduced somewhat. The 'cache effect' was observed in the manual CG implementation, already. It is responsible for the better that linear speedup for data sizes not fitting in one cluster cache. For small data sets this additional bonus going multi-cluster escapes.

Now, let's look at configuration 2. We said before, the global memory bandwidth has a maximum twice as high as configuration 1. We might expect the saturating curves from configuration 1 to continue their ideal behavior up to 4 clusters. That this is not the case reveals an interesting aspect of the global memory system: The shuffle exchange network, connecting global memory and processors, is connected in an ideal way in configuration 1. The speedups are bound by the memory bandwidth only. In Configuration 2 the network was changed in preparation for Cedar's final 32 processor/64MB global memory shape. In the measured intermediate state it represents a considerable bottleneck. The speedup curves are less distinctly 'linear then saturating'. They are flattened more gradually.

Figure 7 exemplifies the impact of the shuffle-exchange network configuration on the system performance. Although we think this is the most interesting lesson to learn, the reader might be interested in the expected behavior of the final Cedar configuration. The global memory will be able to feed the processors at a speed close to the 2-cluster point in configuration 1. The network will have twice as many global memory paths per processor as in configuration 2. Thus, we can expect a better speedup behavior than any of the configurations measured in Figure 7.

6 Automatic vs manual parallelization

Here we compare the explicit parallel implementation of the standard Conjugate Gradient algorithm with the code automatically generated by the Cedar restructurer. We do **not** contrast the compiler techniques with the transformations made for the preconditioned CG and the reduced system approach. These variants need significant changes at the algorithmic level, the automation of which is not a current goal of the Cedar restructurer. The issue of recognizing an algorithmic structure at a high level and replacing it by a better code may belong in the area of 'problem solving environments.' Of course, one can consider the integration of compilers in such environments a desirable future goal.

Tables 2 and 3 give the times for one iteration step obtained for several versions of CG for a large and a small linear system. *Comp 1cl* shows the execution time of the program optimized for 1 cluster. *Comp shared* shows the times for the automatically compiled multi-cluster version using the 'shared-memory' scheme, whereas *Comp dist* shows the 'distributed-memory' compilation results. The next two columns show the timings of the explicit parallel implementations. *CGM* has all data in cluster memory whereas *CGGM* also exploits the global memory. All measures correspond to Cedar configuration 2 of Figure 7.

We can summarize the tables as follows.

- for large data sizes automatic compilation achieves 60% – 90% of the manually gained performance of CGM, depending on the compilation scheme.
- For small data sizes the manual version performs significantly better. This is mainly due to the lower overhead of the synchronization constructs chosen when manually programming.
- Using knowlege of the architecture and the algorithm can yield an additional performance gain of the manual version. In CGGM, storing some of the data in global memory led to a better use of the caches and resulted in an additional speedup of 20% for the 255 data set.
- On one cluster, the *Comp* columns are faster. This is partly caused by the fact that the kernels of CGM and CGGM were compiled by the Alliant FX/8 Fortran compiler which uses a different memory mapping strategy than the Cedar Fortran compiler. Besides, remember that *Comp shared* accesses all and *Comp dist* 50% of the data in global memory, which has a higher bandwidth than one cluster memory.

no. clusters	Comp 1cl	Comp shared	Comp dist	CGM	CGGM
4		85	53	47	38
3		92	65	66	56
2		107	93	108	102
1	192	168	183	226	226

Table 2: Times in milliseconds for one iterationstep for n=255

no. clusters	Comp 1cl	Comp shared	Comp dist	CGM	CGGM
4		6.6	3.9	2.3	2.7
3		7.1	4.8	2.9	3.2
2		7.9	5.7	4.0	4.4
1	8.6	11.6	9.8	7.2	8.0

Table 3: Times in milliseconds for one iterationstep for n=63

6.1 'Shared-memory oriented' compilation

For the standard CG the manual version is about a factor 2 faster than the automatic code, using the 'global-memory scheme'. The main reason for this was identified as the global memory bandwidth which is not sufficient to feed all Cedar processors at full speed. Since we had a near-linear increase in performance going from one to two clusters, we could expect this ideal behavior to continue up to 4 clusters, given a two times higher bandwidth.

One may consider the inverse of the global memory speed as a measure for Cedar's 'distributed system degree': with a high bandwidth the benefit from cluster memories is small, placing Cedar close to a traditional supercomputer architecture, such as Alliant FX/8 or Cray YMP. With a small bandwidth, the cluster memories become the main data carriers, turning the architecture into a distributed memory multiprocessor. Using this terminology, we can attribute the performance limitation, when compiling in a shared-memory oriented way, to Cedar's distributed system degree.

This thought is reinforced when reviewing the parallelization techniques applied in both cases. The manually applied data distribution techniques are opposed to the traditional

shared-memory oriented compilation scheme, which was, as we described, our first restructuring approach. The *Comp dist* columns represent the compiler performance after 'teaching' it about distributed architectures. Still, there are differences between the manual and automatic schemes, which we will discuss next.

6.2 'Distributed-memory compilation'

The parallelism model of the hand-optimized CG, shown in section 4.1, envisions a number of independent tasks, each running on a cluster. They synchronize explicitly when cross-cluster communication is needed. The compilation scheme, on the other hand, produces a single stream of loops, syntactically close to the original sequential program. These loops, then, are executed employing all processors. They all terminate with an implicit barrier synchronizing the whole Cedar complex. Therefore, additional explicit synchronization is not necessary. Communication – for example the collection of partial dotproduct sums – can be done between the loops.

If we look one step further into the implementation we find a scheme even closer to the manual programming. Spread loops (SDOALLS) are implemented using a fast microtasking scheme, starting 'helper tasks' on all clusters at the program start, and using a fast 'awake' and 'sleep' mechanism to employ them as the program runs. These helper tasks are directly comparable to the explicit tasks used in the manual CG. The manually inserted synchronizations correspond to the 'sleep' points at the loop ends. It is to be expected that the manual choice of synchronization functions is smarter than the generic barriers terminating all loops. As mentioned earlier, we could remove some of this overhead by fusing loops, thus removing the barrier completely. This is possible at points 1 and 3 in Figure 5 which correspond to the points that needed no synchronization in the manually translated code.

7 Conclusions

The Conjugate Gradient experiments have given us much insight into the intricacies and the behavior of the Cedar architecture. They have given answers to many Cedar questions from the angle of an important application example. The following paragraphs reflect the outcome of this research we think is most significant.

CG performance on Cedar The experiments presented show that for an important class of algorithms the Cedar architecture can be exploited very efficiently. The resulting speedups are remarkable. We could even demonstrate speedups of more than four, comparing one to four cluster runs. These could be obtained through an efficient use of Cedar's memory hierarchy.

Cedar programming methodology An important goal of the Cedar Application research is to find an answer to the question '*how do programs for Cedar-type architectures differ from traditional parallel programs, and what are the design rules for these programs?*' Programming has always had the flavor of an art which makes it hopeless to learn about these issues without studying implementation details and warming up with many examples. Nevertheless, it seems appropriate to conclude with an abstract of what we experienced when writing programs for the new Cedar machine.

A large variety in available constructs and services to use, is a direct consequence of the highly structured Cedar architecture. For the programmer this means there are more decisions to make, such as where to place data, what access mode to chose, what processors

or clusters to employ, and how to coordinate all activities. This requires both, more planning of an algorithm and more tuning of the implementation.

Perhaps, the major question is how to exploit the memory hierarchy. We have seen that both methods can be appropriate, distributing data onto clusters and accessing data from global memory. The latter may be applied when partitioning data results in a large communication overhead or for balancing the load of memories. To keep the cost of global accesses reasonable one must attempt to use cluster memory simultaneously and prefetch global data in block mode. From the programming language point of view the memory structure often puzzles the programmer. One thing to learn is to abandon the premise that data is as accessible as its identifier by the language scoping rules.

Cedar's cluster structure might be considered a side effect of the memory hierarchy. 'Going multi-cluster' has a price one needs to be aware of. Spreading loops, communicating, and synchronizing across clusters can be considerably slower than within clusters. Paging across clusters can even absorb all program performance. Again, we need a better knowledge of the internal behavior than in traditional 'flat' machines.

Automatic vs manual parallelization Explicit manual parallelization is faster. This is to be expected. Partly this is because manual programming can adapt quicker to new technologies. The hand-optimized CG is tailored to the Cedar architecture, whereas the compiler technology is only about to learn how to automate these transformations. The currently available compiler achieves approximately 50% of the manually gained performance. However, we have also shown a compilation scheme, we are about to implement, yielding a performance close to the manually gained one.

References

[CGM85] P. Concus, G. Golub, and G. Meurant. Block preconditioning for the conjugate gradient method. *SIAM J. Sci. Stat. Comput.*, 6:220–252, 1985.

[EHJP90] R. Eigenmann, J. Hoeflinger, G. Jaxon, and D. Padua. Cedar Fortran and its Compiler. In *CONPAR 90*, 1990.

[Emr85] Perry A. Emrath. Xylem: An Operating System for the Cedar Multiprocessor. *IEEE Software*, 2(4):30–37, July 1985.

[GJG88] Kyle Gallivan, William Jalby, and Dennis Gannon. On the problem of optimizing data transfers for complex memory systems. *Proc. of 1988 Int'l. Conf. on Supercomputing, St. Malo, France*, pages 238–253, July 1988.

[GPHL90] Mark D. Guzzi, David A. Padua, Jay P. Hoeflinger, and Duncan H. Lawrie. Cedar Fortran and other Vector and Parallel Fortran dialects. *Journal of Supercomputing*, pages 37–62, March 1990.

[HMD+86] Christopher Huson, Thomas Macke, James R.B. Davies, Michael J. Wolfe, and Bruce Leasure. The KAP/205: An Advanced Source-to-Source Vectorizer for the Cyber 205 Supercomputer. In Kai Hwang, Steven M. Jacobs, and Earl E. Swartzlander, editors, *Proceedings of the 1986 International Conference on Parallel Processing*, pages 827–832, 1730 Massachusetts Avenue, N.W., Washington D.C, 20036-1903, 1986. IEEE Computer Society Press.

[Kuc88] Kuck&Associates, Inc., Champaign, IL 61820. *KAP User's Guide*, 1988.

[Meu84] G. Meurant. The block preconditioned conjugate gradient algorithm on vector computers. *BIT*, 24:623–633, 1984.

[MS88] U. Meier and A. Sameh. The behavior of conjugate gradient algorithms on a multivector processor with a hierarchical memory. *J. Comp. App. Math.*, 24:13–32, 1988.

[Saa85] Y. Saad. Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM J. Sci. Stat. Comput*, 6:865–881, 1985.

[vdV86] H. van der Vorst. The performance of fortran implementations for preconditioned conjugate gradients on vector computers. *Parallel Computing*, 3:49–58, 1986.