Cidre: Programming with Distributed Shared Arrays

Françoise André, Yves Mahéo

To cite this version:

Françoise André, Yves Mahéo. Cidre: Programming with Distributed Shared Arrays. 3rd International Conference on High-Performance Computing (HiPC '96), Dec 1996, Trivandrum, India. pp.439-444. hal-00496288

HAL Id: hal-00496288
https://hal.science/hal-00496288

Submitted on 30 Jun 2010

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Cidre: Programming with Distributed Shared Arrays

F. André and Y. Mahéo
IRISA, Campus de Beaulieu, F-35042 Rennes, France
Email: fandre@irisa.fr

Abstract

A programming model that is widely approved today for large applications is parallel programming with shared variables. We propose an implementation of shared arrays on distributed memory architectures: it provides the user with an uniform addressing scheme while being efficient thanks to a logical paging technique and optimized communication mechanisms.

1 Introduction

Modularity and extensibility are two strong points of distributed memory architectures. In these machines, that are composed of interconnected processor-memory nodes, the number of nodes can vary easily so that the power of the machine is adapted to the size of the problem and to the expected performances.

Among these architectures one can find on the one hand machines built with parallel computing in mind like the Intel Paragon or the IBM SP2 and on the other hand, high performance networks of workstations like for instance ATM networks of PCs. This second family makes parallel machines available to a large number of users.

Despite important research efforts and a remarkable improvement in the last few years, programming this kind of machines remains complex because of the distribution of the memories that imposes the use of communication operations.

The programming model that is widely approved today for large applications is parallel programming with shared variables. With this model, the programmer can bring out the parallelism that will yield performance while keeping a global view on the manipulated data structures.

Improving the use of distributed memory architectures relies on the implementation of the model of communicating-through-shared-variables processes. Distributed shared memory systems are a solution to this problem [8, 7, 5]. Shared variables are placed in the virtual memory, hence they are addressed in a uniform way. However, this apparent ease of use is tempered with several drawbacks. The page size is independent from that of accessed variables. This may bring about a larger than necessary amount of communication. Above all, when a page contains several variables accessed in parallel by different processors, some "ping-pong" communication patterns may occur.

Our objective is to propose an alternative to this distributed shared memory system. We have designed an implementation of shared variables that is original and efficient on the following two key-points:

- On the addressing side: it provides the user with an uniform addressing scheme for his variables but the interpretation in terms of local addresses in distributed memories is optimized thanks to a logical paging technique.
- On the communication side: we use message passing for required elements but we largely take benefit from vectorization techniques, connected to our logical paging mechanism, in order to reduce the number of messages and the amount of transferred elements.

The remaining of this paper expounds the above-mentioned mechanisms. Section 2 gives a global presentation of the programming environment and details the different levels of abstraction. Through an example, section 3 describes the programming model offered to the user. Section 4 is devoted to the implementation of the distributed array library and explains the addressing techniques and the communication optimizations. Section 5 concludes.

2 Structure of the programming environment

In this paper, we focus on data structure management. Other programming aspects like parallel pro-

<table>
<tr><td>Level 6</td><td colspan="2">High level programming :
provides access to any shared data structure,
including protection and coherence control</td><td></td></tr>
<tr><td>Level 5</td><td colspan="2">Coherence and protection protocols library</td><td></td></tr>
<tr><td>Level 4</td><td>Distributed array library and
coherence function</td><td>Dynamic structures
library (lists, trees)</td><td rowspan="2">CIDRE</td></tr>
<tr><td>Level 3</td><td colspan="2">Logical paging mechanism</td></tr>
<tr><td>Level 2</td><td colspan="2">Communication and threads system</td><td></td></tr>
<tr><td>Level 1</td><td colspan="2">Physical architecture</td><td></td></tr>
</table>

Figure 1: Structure of the programming environment

cess management are not addressed here.

The most external level provides a programming environment where the user can declare any data structure that may be accessed by several processes. References to this shared data structure are made through the usual notation of any programming language. As data structures may be read or written by many processes, the programmer is given different protection mechanisms and coherence control protocols. These tools are typically included in a library. References to several works on such protocols may be found in [5].

The high level programming model is not fixed in our environment : different versions may be proposed depending on the chosen source language.

High level programming layers use the distributed data structure management system described in this paper. The distributed shared array library CIDRE at the 4th level allows the creation of data structures like multi-dimensional arrays and the specification of their logical decomposition upon a distributed memory architecture. This decomposition is expressed in HPF style.

As we said, the usual index notation is used at levels 5 and 6 to access data structures. Indexes are computed globally according to array bounds regardless of the distribution. That is the key point of the programming model we propose : references to distributed arrays are the same as shared memory references. The example 2 illustrates this point.

The CIDRE library provides mechanisms to transform global addresses into physical addresses and performs data communication if necessary. Moreover, a synchronization function called `coherence` allows groups of parallel processes sharing the same data structure to synchronize so that they have the same coherent view of the structure. This function, detailed in section 3, is the key element for the construction of enhanced protocols at the 5th level.

Other libraries may be built at level 4 to handle other data structures than arrays.

The CIDRE library implementation is built upon a mechanism we call logical paging. Level 3 manages the distribution of arrays in different local memories. A distributed array is described as a set of *logical pages*; the size of these pages is related to the size of the array. The processes that own a part of the array in their local memories are given an array descriptor which consists mainly of a table of logical pages. It will be explained in section 4 how we manage to translate references in a very efficient way, which is a major advantage of our model.

Finally, according to the location of the requested page, access may be local or distant. For the distant accesses, we use the communication system of the target machine (level 2). Though it is not compulsory, the use of threads makes the implementation of the distributed arrays library easier, allowing overlap between different activities. At last, level 1 represents the physical layer.

Many shared object libraries have been conceived lately [1, 3, 11, 4]. Situated at different levels of user-interface, they provide various coherence protocols. Compared to these libraries, the CIDRE library offers optimizations for data structure implementation and communication management.

3 The programming model

In this section we describe the library interface and the way it may be used for programming parallel applications.

The appropriate programming model is based on user-defined parallel processes. As it is often the case in the context of highly parallel architectures, processes may execute the same basic code according to the SPMD (Single Program Multiple Data) model. Depending on its identity or on the data it owns, each process will specialize in executing specific parts of the program.

The CIDRE library allows user processes to share variables which will be implicitly distributed according to a user-defined partitioning scheme.

The example in figure 2 illustrates the use of the library. P processes are involved in the execution of the given code which corresponds to a Jacobi algorithm. Shared arrays are declared by calling the CIDRE function `create`. This function takes as parameters the dimensions of the array, followed by the definition of how it is partitioned. Cyclic or block partitioning schemes on the P processors of the architecture are possible. In the example, $N \times N$ matrices A and B are defined and partitioned into blocks of size $N/P \times N$.

Our programming model is specifically defined to allow global references to shared variables, avoiding the need for explicit data transfers and global to local addresses computations. The assignment

$$\texttt{write}(B, i, j,\ f(\texttt{read}(A, i+1, j), \texttt{read}(A, i-1, j), \texttt{read}(A, i, j+1), \texttt{read}(A, i, j-1)))$$

where f is a predefined function, illustrates these facilities.

In the example, process P_i owns the blocks of lines $(N/P \times i)$ to $(N/P \times (i+1) - 1)$. To execute its computation at step k, it needs the line $(N/P \times i - 1)$, after it has been computed by process P_{i-1} at step $k-1$, and the line $(N/P \times (i+1))$ computed by P_{i+1} at step $k-1$. Reciprocally, P_{i-1} needs the line $(N/P \times i)$ and P_{i+1} needs the line $(N/P \times (i+1) - 1)$.

To ensure that each process uses up-to-date values at step k (i.e. values computed at step $k-1$), we introduce two synchronization-and-coherence operations named `coherence`. The first one performs a cooperation between P_i and P_{i-1} for updating their lines $(N/P \times i - 1)$ $(N/P \times i)$ and for providing to both processors a coherent view of these lines; the second one makes P_i et P_{i+1} cooperate for obtaining an up-to-date and coherent view of lines $(N/P \times (i+1) - 1)$ and $(N/P \times (i+1))$.

Generally speaking, the parameters of the `coherence` function describe the set of shared array elements (an array section defined by a lower bound, an upper bound and a step in each dimension) that has to be made equally visible to a group of processes. This group is referenced to by the last parameter. The semantics of the `coherence` function applied to an array section AS and a group of processes G is the following:

- Synchronization of all the processes in G in order to take into account every write operation performed on AS since the last `coherence` call (or the beginning of the execution);
- Broadcasting of an up-to-date version of AS to all processes in G.

Of course, if several writes to the same array element have been performed by different processes before the `coherence` call, the content of the up-to-date version of this element is non deterministic.

4 Logical paging mechanism

The CIDRE library provides access to shared distributed arrays. The involved mechanisms have been used in the HPF Pandore compiler [2]. They exploit logical paging of arrays according to the user-specified rectangular block distribution. The goal is to have a quick elementary access while keeping the memory cost at a reasonable level [10].

The multi-dimensional address space defined for each array is linearized and split into pages. These pages are used to store temporary copies of distant data as well as local data.

Elements are uniformly accessed : global indices are translated into a page number and an offset in the page. The couple (PG, OF) and a table of pages available on each processor are then used to access the corresponding memory element.

The size of the pages and the direction of the pages — *i.e.* a linearization function — are defined for each array according to the array distribution parameters. The direction of the pages corresponds to the dimension of the largest block extent. The page size is chosen to be a power of two to speed up accesses : computation of the couple (PG, OF) only needs simple logical operations (shifts and masks).

Two different cases may occur:

- If there is a non-distributed dimension, the page size is equal to the first power of two greater than

```
process myself
  A = create('A', N, N, N/P, N)
  B = create('B', N, N, N/P, N)

  prev_set = {myself, myself−1}
  next_set = {myself, myself+1}

  my_first_line = N/P*myself
  my_last_line = N/P*(myself+1) −1

  for k=1 to nloop
    if (myself ≠ 0)
      coherence(A, my_first_line-1,my_first_line,1,0,N-1,1, prev_set)
    if (myself ≠ P − 1)
      coherence(A, my_last_line,my_last_line+1,1,0,N-1,1, next_set)
    for i = my_first_line to my_last_line
      for j = 1 to N − 2
        write(B,i,j, f(read(A,i + 1,j), read(A,i − 1,j), read(A,i,j + 1), read(A,i,j − 1)))
    for i = my_first_line to my_last_line
      for j = 1 to N − 2
        write(A,i,j, read(B,i,j))
```

Figure 2: Programming example : Jacobi algorithm

the size of the array in that dimension. Computation of (PG,OF) is then very efficient (identity in the 2D-case).

- If all the dimensions are distributed, the page size is the first power of two lower than the largest block extent. A page may then overlap a block border. In this case, each of the involved processors is responsible for a part of the page.

In addition to efficient elementary accesses, logical paging permits the optimization of the communications involved in the synchronization-and-coherence operation.

Communications are organized in *segments* (adjacent elements of a page). Direct communications are used to transfer big segments without any communication buffer, while small segments are aggregated in a larger buffer to minimize the effect of message latency. The limit between small and big segments can be expressed as a function of platform-specific parameters. Furthermore, multiple occurrences of elements are eliminated when preparing the transfers. A complete description of these mechanisms is available in [9].

5 Performances of logical paging

We have already compared the joint use of the logical paging system discussed above and message passing with shared virtual memory in the framework of the HPF Pandore compiler [9].

Indeed, two versions of the compiler have been written. With the first one, the generated code makes use of the logical paging system and of a portable message passing library, the POM library [6], that allows executions on several parallel architectures and systems (Intel iPSC/2, Intel Paragon, BSD Sockets, PVM...). The second version of the compiler generates code for the SVM Koan [7] build on top of the NX/2 system on the Intel iPSC/2.

Several experiments have been made on the iPSC/2 in order to compare these two approaches. Figure 3 shows the speedups obtained for three numerical kernels: the Jacobi iterative relaxation, the LU factorization and the matrix-matrix product. These results give a good idea of how the CIDRE library would compare to a SVM because, for these regular examples, the code produced by the Pandore compiler is very similar to a hand-coded version of the parallel SPMD code.

Speed of local accesses turns out to be a critical parameter for the overall efficiency. In this respect, logical paging is very close to SVM —that can be

considered optimal— as illustrated in the Jacobi example.

As logical paging is associated with message passing, complex communication patterns can be handled more efficiently than with SVM. This is the case with the LU factorization and the matrix-matrix product where broadcasting of lines and above all communications of parts of lines are necessary. To solve this, broadcasting is employed in both systems. But in the logical paging version, segment broadcasting is carried out so that the number of messages and the amount of transferred elements is kept at a minimum, whereas in the SVM version, a producer-consumers communication pattern for which entire pages are broadcasted is used. As a consequence, a much more important falling off can be observed for the SVM version when the number of processors increases, especially for small array sizes. Moreover, it is clear that the difference would be greater without this broadcasting protocol that is to say when only point to point page transfers would be authorized through page faults solving.

Besides, we believe that the superiority of the logical paging (combined with message passing) that is used in the Cidre library over SVM is likely to be greater in the context of networks of workstations where the message latency is very high.

6 Conclusion

We are currently implementing the Cidre library for shared distributed arrays. The interface language may be subject to some slight modifications, for example concerning the syntax of primitives such as `coherence`.

Moreover, we must work out some implementation mechanisms. For instance, we are experimenting different solutions to efficiently perform the test that must determine if references correspond to data already present or not in the local memory. A `prefetch` operation appears to provide a good way to avoid numerous executions of this test.

At last, the writing of high level coherence protocols is envisaged in order to have enlightenments on the adequacy of Cidre to parallel application programming and on the performances that may be obtained at the user level.

References

[1] J.-M. Adamo. Arch, an object-oriented library for asynchronous and loosely synchronous system programming. Technical Report 228, Cornell Theory Center – Ithica, NY, December 1995.

[2] F. André, M. Le Fur, Y. Mahéo, and J.-L. Pazat. The Pandore Data Parallel Compiler and its Portable Runtime. In *HPCN Europe '95*, number 919 in LNCS, Milan, Italy, May 1995. Springer Verlag.

[3] Thomas Brandes. Adaptor : The distributed array library. Technical report, German National Center for Computer Science (GMD) – St-Augustin, Germany, April 1993.

[4] W.W. Carlson and J.M. Draper. Distributed data access in AC. In *ACM PPOPP Santa Clara, CA, USA*, pages 39–47, 1995.

[5] M.R. Eskicioglu. A Comprehensive Bibliography of Shared Distributed Memory. *ACM Operating Systems Review*, 30(1):71–96, janvier 1996.

[6] F. Guidec and Y. Mahéo. POM: a Virtual Parallel Machine Featuring Observation Mechanisms. In *International Conference on High Performance Computing*, New Delhi, India, December 1995.

[7] Z. Lahjomri and T. Priol. Koan: a Shared Virtual Memory for the iPSC/2 Hypercube. In *2nd Joint International Conference on Vector and Parallel Processing, CONPAR 92 - VAPP V*, number 634 in LNCS, Lyon, France, September 1992. Springer Verlag.

[8] K. Li and R. Shaefer. A Hypercube Shared Virtual Memory System. In *International Conference on Parallel Processing*, University Park, PA, 1989.

[9] Y. Mahéo. *Environnements pour la compilation dirigée par les données : supports d'exécution et expérimentations*. PhD thesis, IFSIC / Université de Rennes I, July 1995.

[10] Y. Mahéo and J.-L. Pazat. Distributed Array Management for HPF Compilers. In *High Performance Computing Symposium*, Montréal, Canada, July 1995.

[11] J. Nieplocha, R.J. Harrisson, and R.J. Littlefield. Global arrays : A portable "shared-memory" programming model for distributed memory computers. In *Supercomputing*, pages 1–9, November 1994.

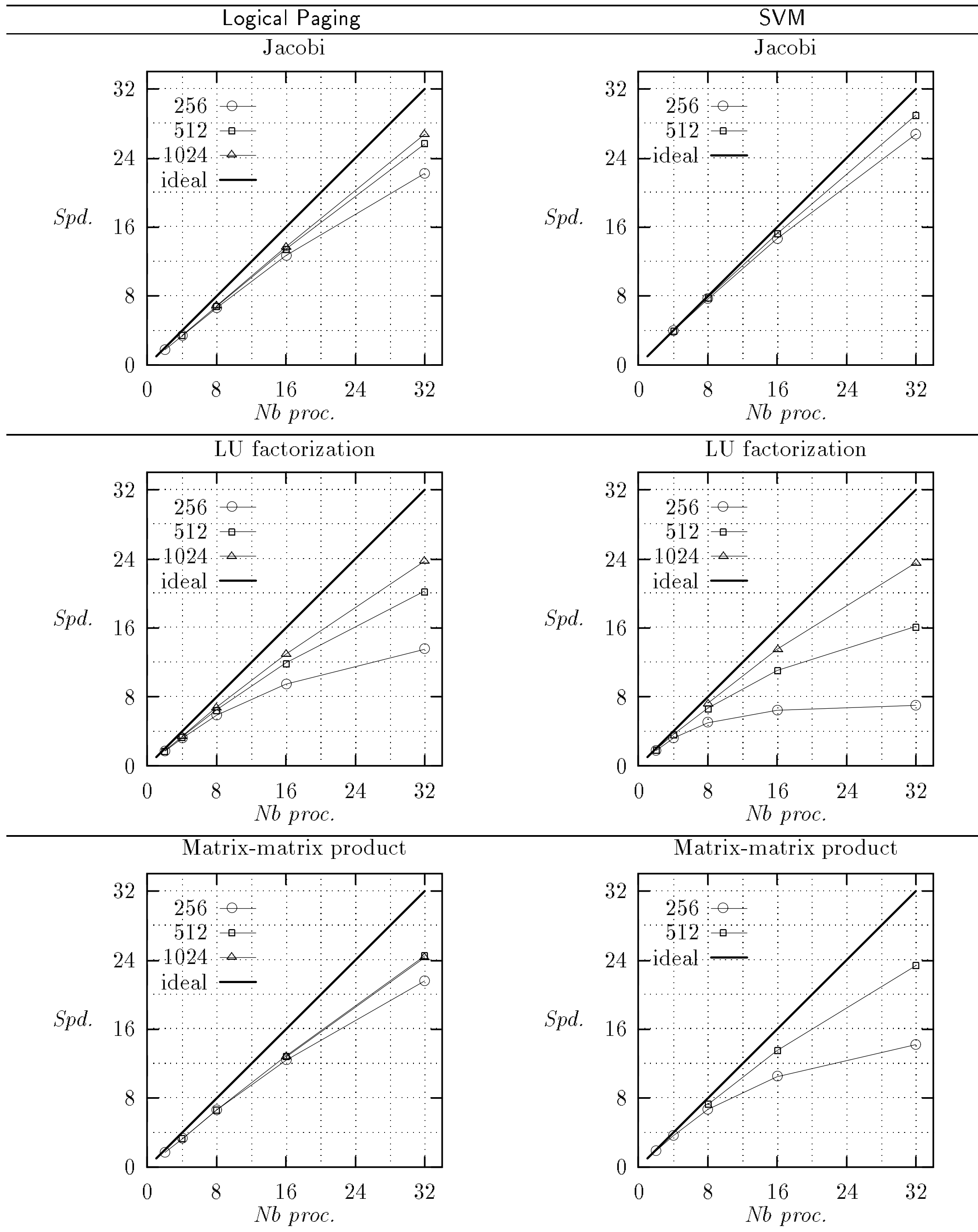

Figure 3: Comparison between logical paging+message passing and SVM