HAL
open science

Overview of the Koan programming environment for the iPSC/2 and performance evaluation of the because test program 2.5.1

François Bodin, Thierry Priol

To cite this version:

François Bodin, Thierry Priol. Overview of the Koan programming environment for the iPSC/2 and performance evaluation of the because test program 2.5.1. [Research Report] RR-1821, INRIA. 1992. inria-00074851

HAL Id: inria-00074851

https://inria.hal.science/inria-00074851

Submitted on 24 May 2006

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.

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France
Tél.: (1) 39 63 55 11

Rapports de Recherche

1992
25ème anniversaire

N° 1821

Programme 1
Architectures parallèles, Bases de données, Réseaux et Systèmes distribués

OVERVIEW OF THE KOAN PROGRAMMING ENVIRONMENT FOR THE iPSC/2 AND PERFORMANCE EVALUATION OF THE BECAUSE TEST PROGRAM 2. 5. 1

François BODIN
Thierry PRIOL

Décembre 1992

RR_1821

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Overview of the KOAN Programming Environment for the iPSC/2 and Performance evaluation of the BECAUSE Test Program 2.5.1 *

Francois Bodin and Thierry Priol
IRISA-INRIA
Campus de Beaulieu
35042 Rennes Cedex

To appear in Proc. of BECAUSE European Workshop
October, 1992
Sophia-Antipolis
France
Publication Interne n°689-Décembre 1992, 16 pages-Programme 1

Abstract

In this paper, we describe a Fortran programming environment using the KOAN Shared Virtual Memory. We then discuss its use for parallelizing a Because benchmark application.

Evaluation de performance de l'application BECAUSE 2.5.1. avec l'environnement de programmation KOAN

Résumé

Nous présentons dans ce papier, un environnement de programmation Fortran utilisant la mémoire virtuelle partagée KOAN. Celui-ci a été utilisé pour paralléliser une application appartenant aux programmes tests Because.

1 Introduction

Since few years, the shared virtual memory (SVM) paradigm has drawn considerable attention. The basic idea of such a concept is to hide the underlying architecture of distributed memory parallel computers (DMPCs) by providing a virtual address space to the user. DMPC could be thus programmed as more conventional shared memory parallel computers. Unfortunately, few experiments have been done to show the effectiveness of SVM on DMPCs.

*This work is partially supported by Intel SSD under contract no. 1 92 C 250 00 31318 01 2

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE (U R A 227) UNIVERSITE DE RENNES I INSA DE RENNES
INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE (UNITE DE RECHERCHE DE RENNES)

The KOAN project has been set up to investigate the use of the SVM paradigm on DMPCs and to check whether this concept is adequate to DMPCs or not. Within this project, several aspects are addressed: SVM design, programming interface, parallel code generation and experiments with parallel algorithms. Earlier results demonstrate that a SVM can be an efficient tool for programming DMPCs. However it does not itself solve the problem of programming DMPCs. For SVM to be successful, it is necessary to find and develop specific techniques for large parallel applications and also to design compiler capable of generating efficient parallel codes. This is the main objective of the KOAN project.

This paper will present a Fortran programming interface, called Fortran-S. It automatically generates parallel codes for the KOAN SVM running on a iPSC/2 hypercube. We outline some of its functionalities on an example from the Because Benchmark Set (BBS.2.5.1).

The paper is organized as follows: Section 2 introduces the concept of shared virtual memory. In section 3, we present briefly the KOAN shared virtual memory. Section 4 gives details of the main features of Fortran-S. Section 6 details the implementation of the benchmarks BBS.2.5.1 using Fortran-S and outlines the performance results obtained.

2 Shared Virtual Memory

A Shared Virtual Memory (SVM) provides to the user an abstraction from an underlying memory architecture [5]. It provides a virtual address space that is shared by a number of processes running on different processors of a distributed memory parallel computer. In order to distribute the virtual address space, the SVM is partitioned into pages[1] which are spread among local processor memories. Each local memory acts as a large software cache for storing pages. A memory management unit (MMU) is needed to provide the user with a linear address by translating virtual addresses to physical ones. An algorithm that implements a shared virtual memory has to solve three problems: *cache coherence*, *page ownership* and *page replacement*. The following sections present some existing solutions for solving these problems and those we chose in implementing KOAN.

2.1 Cache coherence protocol

Since processors may have to read from or to write to the same page, several processors have a copy of a page in their cache. If one processor modifies its copy, other processors run the risk of reading an old copy. A cache coherence protocol is needed to ensure that the shared address space is kept coherent at all times. A memory is considered *coherent* if the value returned by a read from a location of the shared address space is the value of the latest store to that location [1]. A solution is to have either only one copy of a page with write access mode or multiple copies in read-only access mode. The processor that has written most recently into the page is called the *owner* of the page. When a processor needs to write to a page that is not present in its cache or is present in read-only mode, it sends a message

[1] the granularity afforded by hardware virtual memory

to the owner of the page in order to move it to the requesting processor. Then it invalidates all the copies in the system by sending a message to the relevant processors. This strategy is called the *invalidation* approach.

2.2 Page ownership

When a processor needs to access a page, either in write or read access mode, which is not located in its cache, it must ask the owner to send it a copy of the page. This problem is related to the cache coherence protocol described previously. With the invalidation protocol, there is always one owner for a page and the ownership changes according to the page requests coming from other processors. Therefore, the problem is how to locate the current owner of a given page considering that the owner of a page changes. A solution is to update a database that keeps track of the movement of pages in the system. This database can be distributed among the processors as suggested in [5]. Each processor knows exactly the owner of a subset of pages. The subset is fixed by a mapping function that takes the page number as an argument and returns the processor number of the owner.

2.3 Page replacement

The problem of page replacement arises when a processor is the owner of all the pages located in its cache and there is no more free space in the cache. If it requests a new page, it has to find space in its cache. It cannot throw away a page from its cache since it owns all the pages. Moreover it cannot save the pages on external high speed storage devices, like disks, since most of DMPCs do not provide such facilities. Consequently, it has to find a processor which has either a copy of the page or enough space in its cache. If another processor has a copy of the page to be stored, it requires only ownership migration. Otherwise, it requires both page saving and ownership migration. Solutions to this problem are described in [5] and [4].

3 KOAN: a shared virtual memory for the iPSC/2

The KOAN SVM is embedded in the operating system of the iPSC/2. It allows the use of fast and low-level communication primitives as well as a Memory Management Unit (MMU). It differs from SHIVA described in [6] in that it is an operating system based implementation. It appears that the implementation of SHIVA has been done at the user's level without modifying the iPSC/2 operating system and consequently adds some overhead. The KOAN SVM implements the fixed distributed manager algorithm as described in [5] with an invalidation protocol for keeping the shared memory coherent at all times. This algorithm offers a suitable compromise between ease of implementation and efficiency. Let us now summarize some of the functionalities of the KOAN SVM runtime.

3.1 KOAN SVM : a brief overview

KOAN SVM provides to the user several memory management protocol for handling efficiently some particular memory access patterns. One of them may occur when several processors have to write into different locations of the same page. This pattern involves a lot of messages since the page has to move from processor to processor (*ping-pong effect*). At a cost of adding some annotations in the parallel code, we can let them modify concurrently their own copy of a page. Two new constructs: *begin_weak* and *end_weak* which delimit a program section in which a weak cache coherence protocol is used instead of a strong cache coherence protocol. When a *end_weak* is executed, all the copies of a page which have been modified in the weak block are merged into one page that reflects all the changes.

A drawback of shared virtual memory on DMPCs is its inability to run efficiently parallel algorithms that contain a producer/consumer scheme: a page is modified by a processor and then accessed by the other processors. KOAN SVM can manage efficiently this memory access pattern by using the broadcasting facility of the underlying topology of DMPCs (hypercube, 2D-mesh, etc...). All pages that have been modified by the processor in charge of running the producer phase are broadcast to all other processors that will run the consumer phase in parallel. Since the producer has to keep track of all pages that have been modified, two new operating system calls are provided in order to specify both the beginning and the ending of the producer phase.

KOAN SVM provides barrier synchronization as well as subroutines to manage critical sections. These features are implemented by using messages instead of shared variables.

Measuring performance is an important issue since it can be difficult to understand the behavior of programs implemented using a shared shared virtual memory. In order to help the user in this task, KOAN provides both performance analysis and software event tracing.

3.2 Performance evaluation

We have performed measurements in order to determine the cost of various basic operations for both read and write page faults of the KOAN shared virtual memory. For each type of page fault (read or write), we have tested the best and worst possible situation on different numbers of processors. For a 32-processor configuration, the time required to solve a read page fault are in the range of 3.412 *ms* to 3.955 *ms*. For a write page fault, timing results are in the range of 3.447 *ms* to 10.110 *ms* depending on the number of copies that have to be invalidated (as a comparison the shortest message costs 0.3 *ms*).

4 Fortran-S: A Programming Interface to KOAN

Fortran-S mainly aims at providing a convenient programming environment for DMPC. Fortran-S is FORTRAN-77 plus an interface for a shared virtual memory. A first implementation has been made for the Intel hypercube iPSC/2 running KOAN SVM. Parallelization of sequential codes are done using *Cray-Mpp-like* directives [7] (except for data distribu-

`C$ann[Shared(variable)]`	Declare the variable as shared
`C$ann[DoShared(distribution)]`	parallel loops, distribution is the iteration distribution among the processors
`C$ann[BeginSeq()]`	start a sequential section
`C$ann[EndSeq()]`	end a sequential section
`C$ann[BeginCritical()]`	start a critical section
`C$ann[EndCritical()]`	end a critical section
`C$ann[AtomicUpdate()]`	atomic update assignment
`C$ann[WaitDoacross()]`	synchronization for doacross loop
`C$ann[SendDoacross()]`	send synchronization for the doacross
`C$ann[Private(variable)]`	the parameter of a function *variable* is never a shared variable. This limits the amount of code generated by the compiler
`C$ann[WeakCoherency(variable)]`	*variable* is updated at synchronization point
`C$ann[BeginBroadcast(variable)]`	the modification of *variable* will be broadcast
`C$ann[EndBroadcast()]`	broadcast to all processors the modification

Figure 1: Main directives

tion) inserted in the program. The main advantages of using directives is that the parallel program can be compiled for a workstation (sequential emulation of intrinsic are provided). Fortran-S also provides message based primitives that can be used to enhance the efficiency of a program.

In this section we describe the basis of the first fortran-S prototype. Many enhancements are planned, and many of the current limitation should be removed.

4.1 Shared Variables

A variable is declared as shared to the compiler using a directive, the default status for a variable is private. Private variables are duplicated on all processors. A shared variable can only be declared in the main program. They can be passed as a parameter as any other variables. A shared variable is declared using the following directive[2]:

```
      REAL V(N,N)
C$ann[Shared(v)]
```

[2]Name of variables must be lower case in the directives

4.2 SPMD Execution

Parallel execution is achieved using an SPMD execution [2] [10] (Single Program Multiple Data) instead of a Fork-Join model (for instance PCF). At the beginning of the program execution, a thread is created on each processor and each processor starts to execute the program. If no sequential region is declared then all the processors execute the same program.

The work sharing is obtained using shared variables and a parallel do construct. The iterations of the loops are distributed among the processors. Processors are synchronized after each processor has completed its set of iterations. A parallel loop is declared using the directive:

```
C$ann[DoShared("BLOCK")]
        do 5 j=1,m
           do 1 i=1,n
              ini = i-j
              if (i .ne. j) then
                 v(i,j) = sin(abs(ini))
              else
                 v(i,j) = n
              endif
 1         continue
 5      continue
```

The string "BLOCK" indicates the scheduling strategy of the iterations. The strategies are:

1. "BLOCK": chunks of iterations are assigned to processors
2. "CYCLIC": first iteration is affected to the first processors, second to the second processor, and so on.
3. ...

The present distributions are static (i.e. set at compile time), but later versions will include dynamic distributions of iterations (for instance guided self-scheduling [8]). DoShared loop can be nested, but only the iterations of the outermost DoShared loop are considered for parallel execution.

4.3 Sequential Regions

The model also supports sequential region of code that are executed on one single processor. Sequential regions are declared with:

```
C$ann[BeginSeq()]
        print *,'2 : Sequential part...'
        ...........
C$ann[EndSeq()]
```

Sequential regions cannot be nested. A synchronization between all the processors is inserted before the directive

```
C$ann[BeginSeq()]
```

and after the directive

```
C$ann[EndSeq()]
```

It should be noted that a parallel loop cannot appear in a sequential region due to the SPMD execution mode (no thread creation) and vice-versa.

4.4 Weak Coherence and Broadcasting

Weak coherence protocol can be acceded using the following directive (see section 3.1 about the weak coherence protocol):

```
C$ann[WeakCoherency(y)]
```

Where y is a shared variable. For instance in the following loop the variable y is written simultaneously by many processors, so there will be a ping-pong phenomena on pages acceded by more that one processor. The weak coherence protocol removes that phenomena.

```
C$ann[DoShared("BLOCK")]
C$ann[WeakCoherency(y)]
      do 1 i=1,n
         temp = 0.
         do 2 k=ia(i),ia(i+1)-1
            temp = temp + a(k)*x(ja(k))
 2       continue
         y(i) = temp
1     continue
```

The broadcasting facility can be used using the following directives:

```
C$ann[BeginBroadcast(var)]
C$ann[EndBroadcast()]
```

where *var* is a shared variable. For instance, in the following example, the directive `BeginBroadcast(v)` is used to indicate that modification, done on one processor, to the shared array v must be recorded. The directive `EndBroadcast()` indicates that the modifications must be broadcast to all the processors. Implicitly the directives `BeginBroadcast(v)` and `EndBroadcast()` define a sequential code region (as the directives `C$ann[BeginSeq()]` and `C$ann[EndSeq()]`). This strategy is useful since in the parallel loop `do 200 j=i+1,m` the modified part of array v is used by all the iteration of the loop.

```
        do 100 i=1,m
C$ann[BeginBroadcast(v)]
              tmp = 0.0
              do 20 k=1,n
 20              tmp = tmp + v(k,i)*v(k,i)
                 xnorm = 1.0 / sqrt(tmp)
                 do 30 k=1,n
 30                 v(k,i) = v(k,i) * xnorm
C$ann[EndBroadcast()]
C$ann[DoShared("BLOCK")]
         do 200 j=i+1,m
              tmp = 0.0
              do  k=1,n
                tmp = tmp + v(k,i)*v(k,j)
              enddo
              do k=1,n
                 v(k,j) = v(k,j) - tmp*v(k,i)
              enddo
 200     continue
 100  continue
```

4.5 Message Based Primitives and Intrinsic

In many case it is more efficient to used message based primitives instead of shared variables. For instance in order to get a global maximum in the following example, it is more efficient to compute the maximum over each processor then merge the results using the DGLOBALMAX function rather than shared variables to gather local maximums. Most of the message based primitives are provided by the iPSC/2 library. Some of the intrinsic functions are given in table 2.

```
C$ann[Shared(u)]
C$ann[Shared(f)]
C$ann[DoShared("BLOCK")]
        do j = 1, 63
           do i = 3, 128-1, 2
              uold = u(i,2*j+1)
              u(i,2*j+1) = omega * ((2*u(i-1,2*j+1)+2*u(i+1,2*j+1) .... +
     $             omega1*u(i,2*j+1)
              r = DABS(uold - u(i,2*j+1))
              if ( r .gt. rmax ) rmax = r
           enddo
        enddo
        CALL DGLOBALMAX(rmax)
```

MYNODE()	return the processor number
NUMNODES()	number of processors available
GSYNC()	global synchronization primitive
DGLOBALMAX(local max)	compute maximum and broadcast it
DMERGEADD(local vector, len)	add all private vectors and broadcast the result
...	...

Figure 2: Main intrinsic functions

5 Implementation using the Sigma System

The compiler is implemented using the sigma system developed at Indiana University [3]. The system provides support for annotations and program transformations. The organization of the system is shown on figure 3. The new Fortran program makes the call to the KOAN primitives. It also contains synchronization points for updating shared variables in sequential regions. In sequential regions if a reference is made to a shared variable (i.e. requires synchronization for updating) or a private variable (no synchronization) runtime tests are added to decide whether a variable is shared or privates.

6 Programming BBS 2.5.1 with Fortran-S

The benchmark BECAUSE BBS 2.5.1 benchmark program is based on the matrix assembly that occurs in the Everest semiconductor device modeling code. Only the Poisson's equation solver has been used. It consists of a simplified matrix assembly loop over a quasi realistic mesh [9]. The assembly process to parallelize is the following:

```
foreach element in mesh do: main loop to parallelize
        Make local copy of element data: Access to the shared data
        Evaluate divergence contributions : local access
        Add charge contributions to PJACOB, PRHS: local access
        Add PJACOB, PRHS into global array RHS: global data access
endforeach
```

This loop is not parallel due to the write access to the shared array RHS (contribution of all the processors are added to that vector). The contribution of all nodes are summed in that array. The contribution of a node can be added to an element of RHS updated by another processor.

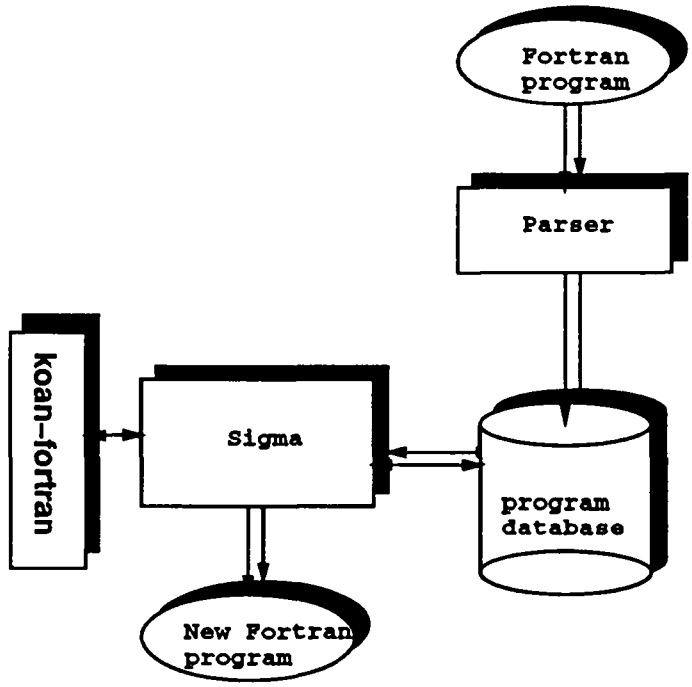

Figure 3: Sigma system organization

6.1 Parallelization

Parallelizing using Fortran-S does not take into account the mesh structure. The mesh arrays are declared as shared variables and so accessible to all processors. Changing the mesh does not modify the code, but in some case may modify load balancing and locality of accesses. The parallelization technique we have applied consists in storing the contribution of a node in a local copy of the RHS vector. The local contributions are then merged into the RHS vector. This particularly means an increase in memory usage because each processor has its own copy of RHS. As a consequence, only the accesses to the shared mesh are done in the parallel loop, all other computations are done on private data. The merge of the data is done after the parallel loop. Figure 4 shows the parallel and sequential part of the program.

7 Preliminary results

In this section we present the results obtained with the first prototype of the compiler Fortran-S. The results for different decomposition factor are given in tables 1, 2 and 3. The table have the following column:

 # proc: Number of processors used

Time (sec.): The time in second

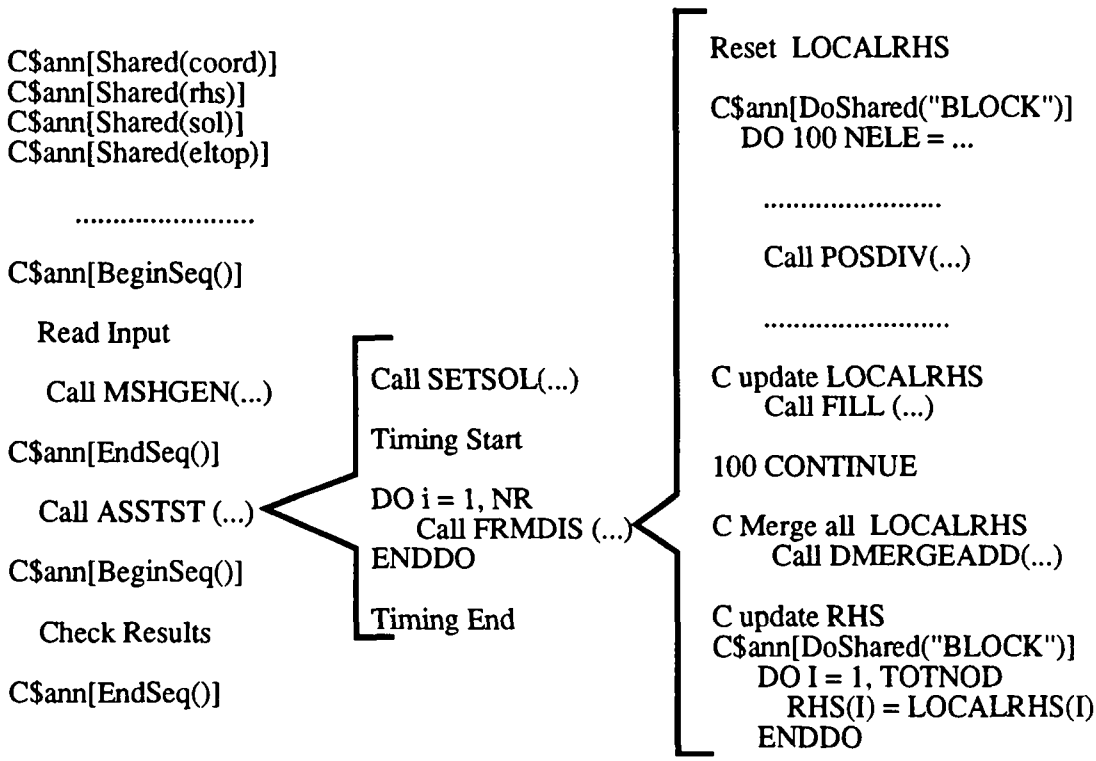

Figure 4: Overview of the program structure

# proc	Time (sec.)	Koan Time (sec.)	Speedup	Mflops	Error
1	33.465	0	1	.095137	.284217E-13
2	16.932	21×10^{-3}	1.97	.188033	.284217E-13
4	8.785	44×10^{-3}	3.80	.362410	.284217E-13
8	4.839	51×10^{-3}	6.91	.660808	.284217E-13
16	2.937	59×10^{-3}	11.39	1.08402	.284217E-13
32	2.130	77×10^{-3}	15.71	1.49473	.284217E-13

Table 1: Benchmark results for a 3D mesh with 10648 nodes

# proc	Time (sec.)	Koan Time (sec.)	Mflops	Error
1	Not enough mem.			
2	32.102	19×10^{-3}	.188262	.941469E-13
4	16.632	40×10^{-3}	.363371	.941469E-13
8	9.073	55×10^{-3}	.666106	.941469E-13
16	5.485	64×10^{-3}	1.10184	.941469E-13
32	3.909	87×10^{-3}	1.54607	.941469E-13

Table 2: Benchmark results for a 3D mesh with 19683 nodes

# proc	Time (sec.)	Koan Time (sec.)	Mflops	Error
1	Not enough mem.			
2	Not enough mem.			
4	28.470	48×10^{-3}	.359840	.230926E-13
8	15.501	62×10^{-3}	.660903	.230926E-13
16	9.319	62×10^{-3}	1.09933	.230926E-13
32	6.579	63×10^{-3}	1.55718	.230926E-13

Table 3: Benchmark results for a 3D mesh with 32768 nodes

Koan Time (sec.): The maximum time per processor spend in KOAN (this includes duration of read fault, write fault, invalidation, ...). Those numbers are provided by the KOAN SVM.

Speedup: $\frac{\text{time for \# proc}}{\text{time with 1 processor}}$

Mflops: Number of Megaflops (in double precision) given in output of the because program. As a comparison a 32 nodes IPSC/2 has a peak performance of approximately 10 Megaflops (in double precision).

Error: The error number given in output of the because program

The text *Not enough mem.* in table 2 and 3 indicates that there was not enough memory to store the mesh. For the speedup computation, it would have been possible to run the program on one processor and allocating more processors to get their memory for shared variables. But in that solution page fault would have occur and so the results would not be completly representative of the performance of one processor.

The SVM KOAN behaves well on that application as shown by the koan time given in table 1, 2 and 3. This thanks to the locality of access to the mesh when using BLOCK iteration partitioning. A non negligible part of the time is spend in the `DMERGEADD()` routine where contributions of the different processor are merge. For instance for a 3D mesh with

10648 nodes and 16 processors the time spend in the DMERGEADD() routine is 0.7 second. This is worst in the case of 32 processors and constitute the actual limit for the speedup.

References

[1] L.M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. on Computers.*, C-27(12):1112–1118, Dec 1978.

[2] F. Darema-Rodgers, V.A. Norton, and G.F. Pfister. *Using A Single-Program-Multiple-Data Computational Model for Parallel Execution of Scientific Applications.* Technical Report RC11552, IBM T.J Watson Research Center, November 1985.

[3] Dennis Gannon, Jenq Kuen Lee, Bruce Shei, Sekhar Sarukaiand Srivinas Narayana, Neelakantan Sundaresan, Daya Atapattu, and François Bodin. Sigma ii: a tool kit for building parallelizing compilers and performance analysis systems. *To appear in* Elsevier, 1992.

[4] Z. Lahjomri and T. Priol. Koan: a shared virtual memory for the ipsc/2 hypercube. In *CONPAR/VAPP92*, September 1992.

[5] Kai Li. *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD thesis, Yale University, September 1986.

[6] Kai Li and Richard Schaefer. A hypercube shared virtual memory system. *Proceedings of the 1989 International Conference on Parallel Processing*, 1:125–131, 1989.

[7] Douglas M. Pase, Tom MacDonald, and Andrew Meltzer. *MPP Fortran Programming Model.* cray research inc. edition, 1992.

[8] C. Polychronopoulos and D. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel computers. *IEEE Transactions on Computers*, 36, December 1987.

[9] Because Esprit Project. *Because Test Programs: BBS.2.5.1 (Matrix Assembly).* 1992.

[10] J.M. Stone. *Nested Parallelism in a Parallel FORTRAN Environment.* Technical Report RC11506, IBM T.J Watson Research Center, November 1985.

8 Appendix: BBS.2.5.1 source

In this section, the modified source code is given.

8.1 Subroutine FRMDIS

```
C$ann[Shared(coord)]
C$ann[Shared(rhs)]
C$ann[Shared(sol)]
C$ann[Shared(eltop)]

...............

      SUBROUTINE FRMDIS(SOL,COORD,TOTNOD,ELTOP,TOTELS,RHS,LOCALRHS)
C
C This subroutine forms the discrete system by looping through all elements
C in the mesh evaluating the local contribution and adding it to the RHS
C vector.
C
C     .. Scalar Arguments ..
      INTEGER TOTELS,TOTNOD
C     ..
C     .. Array Arguments ..
      DOUBLE PRECISION COORD(3,TOTNOD),RHS(TOTNOD),SOL(3,TOTNOD)
      INTEGER ELTOP(10,TOTELS)
      DOUBLE PRECISION LOCALRHS(TOTNOD)
C     ..
C     .. Local Scalars ..
      DOUBLE PRECISION EPSR
      INTEGER I,J,NELE,NODEL
C     ..
C     .. Local Arrays ..
      DOUBLE PRECISION AREAS(6),ELAREA(6),ELCORD(3,8),ELLENG(6),
     +                 ELSOL(3,8),ELVOL(8),LENGTH(6),PJACOB(8,8),
     +                 PRHS(8),VOLS(4)
      INTEGER STEER(8)
C     ..
C     .. Data statements ..
      DATA AREAS/6*0.0625/,LENGTH/6*1D0/,VOLS/4*0.05D0/
C     ..
C
      EPSR = 12.3D0
C
C Initialize LOCALRHS the array used to gather contribution of node
C this is local, results are merged later
C
      DO 9 I = 1,TOTNOD
        LOCALRHS(I) = 0D0
 9    CONTINUE
C
C
C *** LOOP AROUND ALL ELEMENTS
C
C
C$ann[DoShared("BLOCK")]
      DO 100 NELE = 1,TOTELS
C
         NODEL = ELTOP(1,NELE)
C
         DO 30 I = 1,NODEL
           STEER(I) = ELTOP(I+1,NELE)
           DO 20 J = 1,3
              ELSOL(J,I) = SOL(J,STEER(I))
              ELCORD(J,I) = COORD(J,STEER(I))
  20        CONTINUE
  30     CONTINUE
C
         DO 50 I = 1,NODEL
           PRHS(I) = 0D0
           DO 40 J = 1,NODEL
              PJACOB(I,J) = 0D0
  40        CONTINUE
  50     CONTINUE
C
         IF (NODEL.EQ.4) THEN
           DO 60 I = 1,6
              ELLENG(I) = LENGTH(I)
              ELAREA(I) = AREAS(I)
  60        CONTINUE
           DO 70 I = 1,4
              ELVOL(I) = VOLS(I)
```

```
 70       CONTINUE
      ELSE
         ELLENG(1) = ELCORD(1,2) - ELCORD(1,1)
         ELLENG(2) = ELCORD(2,4) - ELCORD(2,1)
         ELLENG(3) = ELCORD(3,5) - ELCORD(3,1)
         ELAREA(1) = 0.25*ELLENG(2)*ELLENG(3)
         ELAREA(2) = 0.25*ELLENG(3)*ELLENG(1)
         ELAREA(3) = 0.25*ELLENG(1)*ELLENG(2)
         ELVOL(1) = 0.125*ELLENG(1)*ELLENG(2)*ELLENG(3)
         DO 80 I = 2,8
            ELVOL(I) = ELVOL(1)
 80       CONTINUE
      END IF
C
      CALL POSDIV(ELSOL,EPSR,ELAREA,ELLENG,PRHS,PJACOB,NODEL)
C
      DO 90 I = 1,NODEL
         PRHS(I) = PRHS(I) + ELVOL(I)* (ELSOL(3,I)-ELSOL(2,I))
         PJACOB(I,I) = PJACOB(I,I) +
     +               ELVOL(I)* (ELSOL(2,I)+ELSOL(3,I))
 90    CONTINUE
C
      CALL FILL(PRHS,PJACOB,RHS,TOTNOD,NODEL,STEER,LOCALRHS)
C
 100 CONTINUE
C
C Merge all contributions
C
    CALL DMERGEADD(LOCALRHS,TOTNOD)
C
C$ann[DoShared("BLOCK")]
    DO  I = 1,TOTNOD
      RHS(I) = LOCALRHS(I)
    ENDDO
C
    END
```

8.2 Subroutine FILL

```
C$ann[NoSideEffect()]
    SUBROUTINE FILL(PRHS,PJACOB,RHS,TOTNOD,NODEL,STEER,LOCALRHS)
C
C Transfer element contribution to RHS vector. Also add element contribution
C to system matrix into the RHS vector (simplification of problem, omits
C the sparse matrix addressing).
C
C    .. Scalar Arguments ..
    INTEGER NODEL,TOTNOD,ITMP,MAXTMP
C    ..
C    .. Array Arguments ..

    DOUBLE PRECISION PJACOB(8,8),PRHS(8),RHS(*)
    INTEGER STEER(8)
    DOUBLE PRECISION LOCALRHS(TOTNOD)
C    ..
C    .. Local Scalars ..
    INTEGER I,INOD,J
    DOUBLE PRECISION RHSTMP
C
    DO I = 1,NODEL
      INOD = STEER(I)
      RHSTMP = PRHS(I)
C
      DO J = 1,NODEL
        RHSTMP = RHSTMP + PJACOB(I,J)
      ENDDO
      LOCALRHS(INOD) = LOCALRHS(INOD) +  RHSTMP
    ENDDO
C
    END
```

LISTE DES DERNIERES PUBLICATIONS INTERNES PARUES A L'IRISA

PI 680 BRANCHING BISIMULATION FOR CONTEXT-FREE PROCESSES
Didier CAUCAL, Dung HUYNH, Lu TIAN
Octobre 1992, 36 pages.

PI 681 DEADLOCK MODELS AND GENERAL ALGORITHM FOR DISTRIBUTED DEADLOCK DETECTION
Jerzy BRZEZINSKI, Jean-Michel HELARY, Michel RAYNAL
Octobre 1992, 26 pages.

PI 683 TARGET TRACKING BY VISUAL SERVOING
Aristide S. SANTOS, François CHAUMETTE
Octobre 1992, 50 pages.

PI 684 UNE DESCRIPTION LINEAIRE COMPLETE ET IRREDONDANTE DU POLYTOPE ASSOCIE AU PROBLEME DU VOYAGEUR DE COMMERCE ASYMETRIQUE A 6 SOMMETS
Reinhardt EULER, Hervé LE VERGE
Octobre 1992, 30 pages.

PI 685 MISE EN CORRESPONDANCE DE SEGMENTS DANS UNE SEQUENCE D'IMAGES PAR UNE APPROCHE LOCALE
Samia BOUKIR, Patick BOUTHEMY, François CHAUMETTE, Didier JUVIN
Octobre 1992, 30 pages.

PI 686 FROM EQUATIONS TO HARDWARE. TOWARDS THE SYSTEMATIC MAPPING OF ALGORITHMS ONTO PARALLEL ARCHITECTURES
François CHAROT, Patrice FRISON, Eric GAUTRIN, Dominique LAVENIER, Patrice QUINTON, Charles WAGNER
Octobre 1992, 18 pages.

PI 687 THE COMPILATION OF PROLOG and its Execution with MALI
Pascal BRISSET, Olivier RIDOUX
Novembre 1992, 90 pages.

PI 688 GENERALISATION DE L'ANALYSE FACTORIELLE MULTIPLE A L'ETUDE DES TABLEAUX DE FREQUENCE ET COMPARAISON AVEC L'ANALYSE CANONIQUE DES CORRESPONDANCES
Lila ABDESSEMED, Brigitte ESCOFIER
Novembre 1992, 34 pages.

PI 689 OVERVIEW OF THE KOAN PROGRAMMING ENVIRONMENT FOR THE iPSC/2 AND PERFORMANCE EVALUATION OF THE BECAUSE TEST PROGRAM 2.5.1..
François BODIN, Thierry PRIOL
Décembre 1992, 16 pages.

ISSN 0249 - 6399