HAL
archives-ouvertes.fr

Recherche des éléments propres d'une matrice symétrique sur une architecture parallèle

Bernard Philippe, Michèle Raphalen

▶ To cite this version:

Bernard Philippe, Michèle Raphalen. Recherche des éléments propres d'une matrice symétrique sur une architecture parallèle. [Rapport de recherche] RR-0662, INRIA. 1987. inria-00075891

HAL Id: inria-00075891
https://hal.inria.fr/inria-00075891

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

N° 662

RECHERCHE DES ÉLÉMENTS PROPRES D'UNE MATRICE SYMÉTRIQUE SUR UNE ARCHITECTURE PARALLÈLE

Bernard PHILIPPE
Michèle RAPHALEN

Mai 1987

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex: UNIRISA 950 473 F
Télécopie : 99 38 38 32

RR n° 662

Publication Interne n° 357

Avril 1987 - 84 Pages

Recherche des éléments propres d'une matrice symétrique sur une architecture parallèle (SIMD / SPMD)

Bernard PHILIPPE

Michèle RAPHALEN

RESUME :

On adapte ici un sous-ensemble de la bibliothèque EISPACK à une architecture parallèle de type SIMD / SPMD.
Les algorithmes correspondants, exprimés dans le langage HELLENA, sont testés sur un simulateur de l'architecture cible.
Cette étude permet, à la fois, de mesurer le parallélisme des programmes obtenus et de juger des avantages et des limites du mode SPMD.

Eigenvalues of a symmetric matrix on a parallel architecture (SIMD/SPMD)

ABSTRACT :

Some procedures, selected from the EISPACK library, have been adapted to a SIMD / SPMD architecture.
The algorithms, coded in HELLENA, have been tested on a simulator of the target architecture.
From this work, on one hand we were able to evaluate how parallel were the obtained programs and on the other hand we listed some advantages and disadvantages of the SPMD mode.

MOTS CLES :

valeurs propres, calcul parallèle, HELLENA.

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

TABLE DES MATIERES

INTRODUCTION

1 - Expression du parallèlisme pour différents types d'architectures

2 - Description de l'ensemble architecture et langage

2.1 L'architecture OPSILA

2.1.1. principes généraux

2.1.2. le simulateur OPSILA de l'IRISA

2.2 Le langage HELLENA - son compilateur

2.3 Remarques sur l'implantation des données en SPMD

3 - Algorithmes diagonalisant une matrice symétrique - parallèlisme

4 - TRED1 - VECT et TRED2 - VECT : versions parallèles de TRED1 et TRED2

4.1 Principes de l'algorithme

4.2 Implémentation

4.3 Résultats des simulations

4.3.1. algorithme TRED1-VECT

4.3.2. algorithme TRED2-VECT

4.4 Commentaires des résultats

5 - TQL1-VECT et TQL2-VECT : versions parallèles de TQL1 et TQL2

5.1 Principes de l'algorithme

5.1.1. calcul des valeurs propres

5.1.2. calcul des vecteurs propres

5.2 Implémentation

5.2.1. calcul des valeurs propres par l'algorithme TQL2-VECT.Simd

5.2.2. calcul des vecteurs propres

5.2.3. schéma général de l'algorithme TQL2-VECT

5.3 Résultats des simulations

5.3.1. algorithme TQL1-VECT

5.3.2. algorithme TQL2-VECT

5.4 Commentaires des résultats

6 - TREPS : version parallèle de BISECT

6.1 Principes de l'algorithme

6.2 Implémentation de TREPS en modes SIMD et SPMD

6.3 Résultats des simulations

6.4 Commentaires des résultats

7 - Discussion

7.1 Cas où seules les valeurs propres sont requises

7.2 Cas où des vecteurs propres sont requis

7.3 Deux situations entrainant des dégradations

7.4 Pratique du SPMD

PAPIER RECUPÉRÉ ET RECYCLÉ

CONCLUSION

Annexe 1 : Table des temps d'exécution des instructions du simulateur
Annexe 2 : Programme TRED
Annexe 3 : Programme TQL-VECT
Annexe 4 : Programme TREPS
Annexe 5 : Définition des matrices de test.

Cette étude se situe dans le cadre de la convention 004/41/85 entre le Centre d'Electronique de l'Armement (CELAR) et l'INRIA/IRISA.

INTRODUCTION

Le domaine de l'étude consiste à évaluer différents types d'algorithmes parallèles permettant de diagonaliser une matrice réelle symétrique. Cette évaluation est faite sur la base d'une architecture de type OPSILA, les programmes étant écrits en langage HELLENA. Les performances sont mesurées sur un simulateur de l'architecture cible.

L'étude a un double but :

- D'une part, pour l'architecture donnée, déterminer les algorithmes les plus performants.
- D'autre part, évaluer l'adéquation de l'architecture au problème posé et apprécier l'effort de mise en oeuvre.

Une attention spéciale est portée sur le SPMD, qui est une caractéristique originale de l'architecture.

1 - Expression du parallèlisme pour différents types d'architectures

Dans cette étude nous nous réfèrerons à trois types d'architectures que nous définissons brièvement maintenant.

Une *architecture SIMD* (Single Instruction stream Multiple Data stream) comporte un certain nombre de processeurs synchrones. Ils sont commandés par une seule unité et exécutent donc simultanément les mêmes instructions. Certains processeurs peuvent être "masqués" pour qu'ils n'exécutent pas l'instruction. Ces architectures permettent ainsi d'exécuter des *instructions vectorielles*.
Habituellement un vecteur est une suite d'éléments dont les adresses en mémoire

sont en progression arithmétique. La longueur des vecteurs étant en général supérieure au nombre de processeurs, il est nécessaire de décomposer ces vecteurs en tranches avec l'une d'entre-elles qui peut être incomplète. Lorsque l'instruction vectorielle a plusieurs opérandes, il est nécessaire d'aligner les tranches pour que les composantes correspondantes se situent dans le même processeur. Pour réaliser cet alignement, il existe un réseau qui relie la mémoire aux processeurs ; ce réseau synchrone est aussi dirigé par l'unité de commande.

A l'opposé de cette situation totalement synchrone, il existe les *architectures MIMD* (Multiple Instruction stream Multiple Data stream). Ici, les processeurs sont capables d'exécuter leur programme, ils sont donc asynchrones. Si on suppose qu'il existe encore une mémoire globale que tous les processeurs peuvent atteindre, l'architecture est dite *fortement couplée* ; en effet les processeurs peuvent alors échanger de l'information à travers cette mémoire commune. Pour permettre ces échanges il est nécessaire d'avoir des primitives de synchronisation comme l'instruction indivisible du test-and-set. Ce type d'architecture est donc le plus riche de tous. Il permet entre autres d'exécuter des boucles où deux itérations successives peuvent partiellement se recouvrir mais où certaines synchronisations sont nécessaires. Par exemple supposons que l'on ait un programme du type :

```
A(0) := ...
pour i := 1 jusqu_a n
   boucle
      ...
      ...
      A(i) := ... A(i-1) ...
   fin_boucle.
```

Chaque itération peut s'exécuter indépendamment des autres itérations sauf pour la dernière instruction qui doit attendre l'exécution de l'itération précédente. On en déduit donc une *exécution "pipelinée" de la boucle.*

Entre ces deux architectures, on en considèrera une troisième que l'on peut voir comme un compromis : le SPMD (Single Program Multiple Data stream). Ce concept a été introduit pour l'architecture OPSILA[Au 85] et sera étudié en détail dans la suite de cette étude. Partant d'une *architecture SIMD* où la mémoire est divisée en autant de bancs que de processeurs, on suppose qu'à certains moments les processeurs peuvent se désynchroniser et exécuter un programme qui

est situé avec ses données dans le banc qui lui correspond. Cette architecture permet donc d'exécuter de manière asynchrone une *boucle pour-tout* (ou *doall*) dont les itérations sont indépendantes mais dont le chemin d'exécution est différent. Ce compromis est donc un assouplissement du SIMD pur qui ne peut, au moins de manière efficace, traiter de telles boucles, mais il est aussi une forte restriction du MIMD qui permet, lui, l'échange de données entre processeurs.

Dans la présente étude, nous nous limiterons donc à la recherche d'instructions vectorielles ou de boucles *pour-tout* puisque l'architecture visée est une architecture SIMD/SPMD. Elle est décrite dans le paragraphe suivant.

2 - Description de l'ensemble architecture et langage

2.1. : L'architecture OPSILA

2.1.1 : principes généraux

Par hypothèse dans cette étude, l'architecture ciblée est de type OPSILA, c'est à dire du type du calculateur réalisé au Lassy[Au 85] . Elle est aussi celle qui est décrite dans le simulateur de l'IRISA. Le principe de ce type d'architecture est décrit maintenant.

Le calculateur se compose de deux parties (Figure 2-1)

- *l'unité de gestion* (UG) constituée d'un processeur et d'une mémoire ; cette unité effectue tous les calculs scalaires et prépare les instructions vectorielles.

- *le processeur parallèle* constitué de nproc processeurs a priori synchrones (nproc = 4,8,16) et du même nombre de bancs de mémoire. La mémoire est reliée aux processeurs par un réseau d'interconnexion (de type omega) synchrone. Les processeurs et le réseau sont commandés par une *unité de commande vectorielle* (UCV).

La communication entre les unités de gestion et vectorielle se fait par l'intermédiaire de files d'attente permettant un fonctionnement asynchrone des deux unités.

On peut donc maintenant décrire le mode usuel de fonctionnement de cette architecture, le *mode SIMD* :
quand une instruction vectorielle est rencontrée, l'unité de gestion prépare les

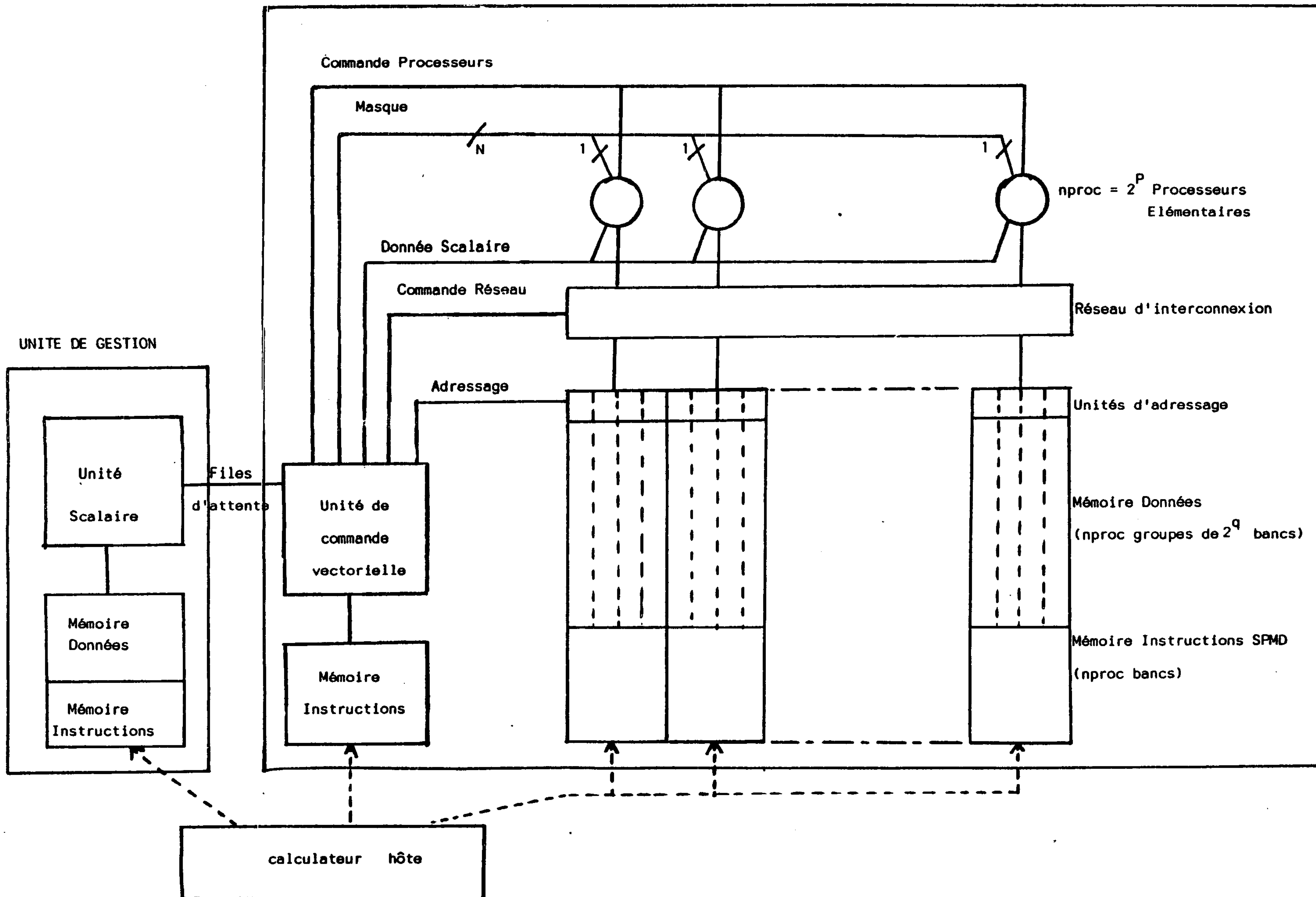


Figure 2.1 : Architecture de type OPSILA.

descripteurs des vecteurs opérandes et résultat ainsi que le descripteur de l'instruction elle-même. Par définition, un vecteur est une suite d'éléments de la mémoire vectorielle dont les adresses sont de la forme :

$$@V + i * r_{int} + j * r_{ext}$$

adresse de base ; raison interne ; raison externe — i,j décrivant des intervalles d'entiers

Une instruction vectorielle a la forme :

<u>pour-tout</u> j <u>dans</u> [1, l_{ext}] ,

<u>pour-tout</u> i <u>dans</u> [1, l_{int}]:

liste d'instructions sur des éléments de vecteurs

<u>fin-pour-tout</u> ;

La *boucle interne* est automatiquement tronçonnée en tranches de nproc itérations par le hardware et le réseau est aussi automatiquement commandé pour permettre l'alignement des composantes correspondantes des opérandes. La *boucle externe* est séquentielle ; sa présence permet de diminuer la circulation entre l'UG et l'UCV, pour les opérations matricielles.

A côté de ce mode SIMD, il en existe un autre : le *mode SPMD*. Ce mode de fonctionnement rend originale l'architecture OPSILA. On suppose maintenant que l'UCV a la possibilité de donner à chaque processeur un ordre permettant d'exécuter un programme qui se trouve dans le banc de la mémoire qui lui est associé. Dans ce cas le réseau n'intervient plus et les processeurs sont asynchrones. Lorsque tous les processeurs sont en attente, l'UCV reprend la main pour retourner au mode SIMD. On considèrera donc une séquence de code SPMD comme une instruction vectorielle particulière.

Une maquette du calculateur OPSILA est construite au LASSY [Au 85].

Le compilateur HELLENA aura un générateur de code pour cette maquette.

2.1.2 : le simulateur OPSILA de l'IRISA

La conception du simulateur correspond à l'architecture du calculateur OPSILA. Le schéma de la figure 2.1. décrit son organisation générale.

Le simulateur est constitué de deux parties asynchrones :
- l'unité de gestion (UG)
- le processeur parallèle (PP), qui présente deux modes de fonctionnement : vectoriel (SIMD) et parallèle (SPMD).

Il permet d'exécuter du code généré par la compilation d'un programme P écrit en langage HELLENA, décrit au paragraphe 2.2. . Ce code (UG, UCV et SPMD) est chargé dans les différentes mémoires d'instructions lors du lancement du simulateur.

Pour pouvoir mesurer l'impact de l'architecture sur le comportement des programmes, en particulier sur leur durée d'exécution, le simulateur est paramétré par des caractéristiques telles que :
- nombre de processeurs élémentaires
- nombre de bancs mémoire parallèle de données
- tailles des files d'attente d'échange entre l'UG et l'UCV
- durée d'exécution des instructions
  . scalaires (UG)
  . vectorielles (SIMD)
  . parallèles (SPMD).

Ces instructions et les durées d'exécution utilisées pour les simulations sont décrites en annexe 1.

2.2. : Le langage HELLENA - Son compilateur

HELLENA [Je 86] est un langage séquentiel classique de la famille PASCAL étendu par des instructions vectorielles, parallèles et SPMD.

L'instruction parallèle de base est le *pour_tout* . Cette instruction se présente comme une instruction d'itération *pour* dont toutes les itérations seraient déclarées indépendantes. Elle permet d'exploiter les niveaux de parallélisme de type pipeline et SIMD des supercalculateurs.

Par exemple dans la procédure MUSECT du programme TREPS (Annexe 4)

on trouve l'instruction :

```
pour_tout k dans [1, neval] :
      U(k) := D(i) - V(k) - (si V(k) = zero  alors absolu E(i)/epsilon
                                               sinon E2(i)/U(k)
                             fin-si) ;
      si (U(k) < zero)  alors S(k) := S(k) + 1 ; fin-si ;
fin_pour_tout ;
```

L'instruction SPMD exprime l'exécution en parallèle d'un code sur des ensembles de données disjoints. Aucune synchronisation ni aucun échange d'information n'est permis entre des tâches s'exécutant dans ce mode. La fin d'exécution des tâches SPMD provoque une synchronisation des processeurs. Il est à remarquer que la largeur de parallélisme est majorée par une constante (le nombre de processeurs) définie dans l'implémentation, et que les communications ne peuvent se faire qu'en retournant au mode SIMD.

Les instructions vectorielles d'HELLENA sont issues d'une extension automatique du champ d'application des opérateurs scalaires du langage aux tableaux de scalaires, comme cela sera fait dans FORTRAN 8X. On a donc deux manières d'exprimer des opérations sur des vecteurs : par une instruction vectorielle ou par une instruction *pour_tout*.

Le champ d'application des instructions vectorielles est étendu par la notion de *patron d'accès* . L'utilisation d'un tel patron permet de manipuler des sous-ensembles de tableaux, comme la diagonale, une ligne, une colonne, ... Par exemple la déclaration :

```
patron ligne de tableau [1,1000] * [1,100]  de reel
         dans tableau [1,1000] de tableau [1,100] de reel :
      pour_tout i dans [1,1000] :
            pour_tout j dans [1,100] : (i,j) fin_pour_tout
      fin_pour_tout ;
```

permet de définir le patron ligne qui sera paramétré par le numéro i de la ligne que l'on désire référencer et qui sera applicable à toute matrice A de type tableau [1,1000]*[1,100] de reel par l'application :

A.ligne (i)

L'élément A(i,j) sera aussi A.ligne (i) (j).

Un des principaux intérêts des patrons réside dans leur efficacité, puisque la plupart des calculs d'indices qu'ils entrainent sont résolus à la compilation.

Le compilateur d'HELLENA est composé de trois programmes correspondant aux étapes suivantes :

- analyse syntaxique
- analyse sémantique
- génération de code.

Après la première et la deuxième étape, il est possible d'appeler un "décompilateur" qui restitue le programme tel que l'analyse l'a compris, possibilité qui s'avère très utile dans la phase de mise au point. La génération de code fournit trois fichiers comportant respectivement le code séquentiel, le code vectoriel et le code SPMD. Ces codes sont alors directement exécutables par le simulateur.

2.3. : Remarques sur l'implantation des données en SPMD

De manière générale un tableau déclaré en SIMD est rangé en mémoire suivant le rangement FORTRAN à partir d'une adresse quelconque, deux éléments consécutifs étant rangés dans deux bancs consécutifs. Par contre en SPMD, les adresses sont des adresses à l'intérieur des bancs. Par exemple la déclaration :

variable D : tableau [1,5] de reel ;
spmd variable E : tableau [1,5] de reel ; fin-spmd ;

provoquera le rangement décrit dans le tableau 2.2. Les variables SPMD peuvent être accédées en SIMD par le préfixe : *en_spmd*
qui considère que la variable obtenue est un tableau[0,nproc- 1] du type de la variable SPMD.

0	1	2	3
	D(1)	D(2)	D(3)
D(4)	D(5)		
E(1)	E(1)	E(1)	E(1)
E(2)	E(2)	E(2)	E(2)
E(3)	E(3)	E(3)	E(3)
E(4)	E(4)	E(4)	E(4)
E(5)	E(5)	E(5)	E(5)

Tableau 2.2 : Exemples de rangements SIMD et SPMD.

Lorsque des *données doivent être utilisées (en lecture) par tous les processeurs en SPMD*, il est nécessaire de les diffuser dans tous les bancs. Par exemple la procédure suivante permet de diffuser le tableau de réel T au tableau de tableau EXT :

```
procedure DIFFUSER-R (valeur n : entier ;
                             T : tableau [1,n] de reel ;
          variable EXT : tableau [0,nproc-1] de
                                 tableau [1,n] de reel ; ) :
     debut
        pour_tout j dans [1,n]
        pour_tout k dans [0, nproc-1]
           EXT(k) (j) := T(j) ;
        fin_pour_tout ;
     fin ;
```

Ainsi l'appel

DIFFUSER-R(5 , D, en_spmd` E) ;

provoquera la diffusion du tableau SIMD dans tous les bancs (D et E sont les tableaux définis au début du paragraphe). On peut remarquer qu'il est possible de conserver le même nom pour un tableau SIMD et son correspondant SPMD (D et E dans notre exemple) puisqu'ils ne sont pas définis dans le même environnement ; ils représentent malgré tout des espaces-mémoire disjoints.

Il reste à résoudre *comment répartir les tâches indépendantes d'un doall* de longueur quelconque sur les processeurs. Pour cela on suppose que le *doall* s'écrit :

```
pour_tout i dans [1,n] :
      tache (i)
fin_pour_tout ;
```

(Cette écriture est symbolique ; elle n'est pas admise en HELLENA).

Supposons que tache(i) n'utilise que les i^{èmes} composantes des vecteurs A1, A2, ... de longueur n. Il est donc nécessaire que toutes ces composantes se trouvent dans le banc correspondant au processeur qui va exécuter cette tâche. Pour aligner tous les vecteurs on procède de la manière suivante :
Si A représente l'un quelconque de ces vecteurs on commence par déclarer un vecteur A en SPMD de longueur $m = \lceil n/nproc \rceil$. Puis on définit le vecteur A en SIMD comme étant la réunion de ces vecteurs SPMD par l'identification suivante :

```
A : pour_tout i dans [1, m * nproc] :
        en_spmd` A (i-1) (1)
    fin_pour_tout ;
```

Ce procédé correspond en fait à un débordement de tableau ; il permet d'identifier un tableau du SIMD avec un tableau de SPMD de même nombre de dimensions. Chaque processeur aura alors à effectuer en SPMD une boucle de longueur m ou m-1. Pour décider entre les deux valeurs, les processeurs doivent connaître leur numéro ; pour cela on peut le faire par :

```
pour_tout k dans [0, nproc - 1] :
     en_spmd` numproc(k) := k ;
fin_pour_tout ;
```

où numproc est une variable entière déclarée en SPMD.

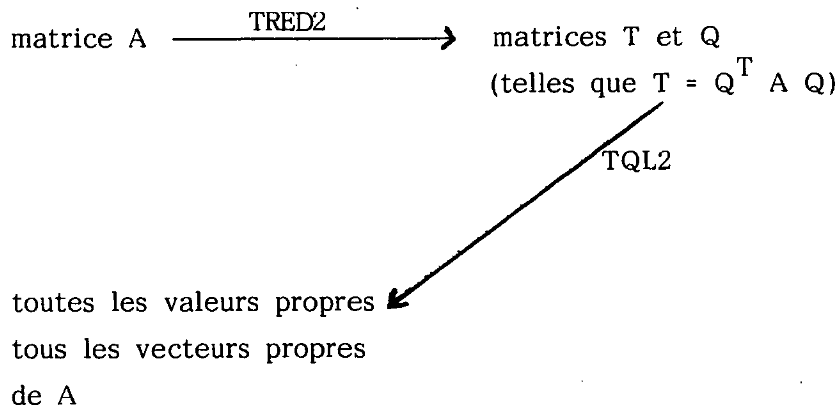
3 - Algorithmes diagonalisant une matrice symétrique - Parallélisme

On se place ici dans le cas d'une matrice pleine ce qui exclut les méthodes de sous espaces qui sont généralement réservées aux matrices creuses car elles ne transforment pas la matrice de départ.

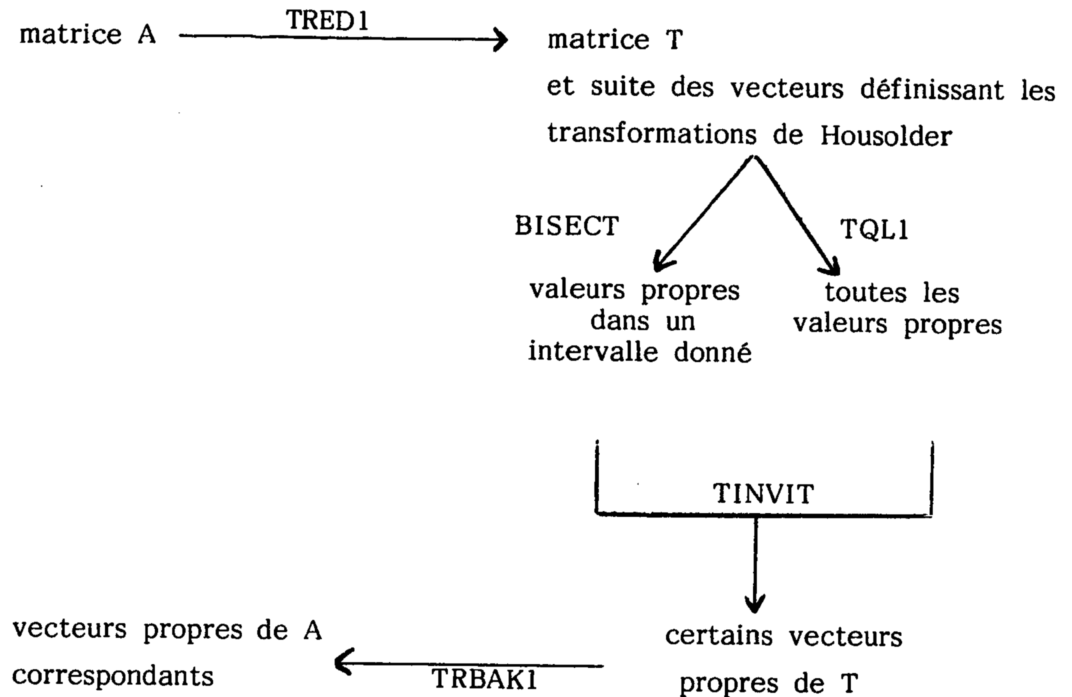
Historiquement, la plus ancienne des méthodes séquentielles est celle de *Jacobi* : elle consiste à construire, à partir de la matrice initiale A d'ordre n, une suite de matrices obtenues par transformations orthogonales (rotations de Givens) qui tend vers une matrice diagonale D, ensemble des valeurs propres de A. Chaque rotation fait apparaitre un zéro (et son symétrique) hors diagonal ; le choix de l'élément à annuler dépend de la politique choisie : on peut par exemple choisir l'élément hors-diagonal de plus grande valeur absolue.

Cette méthode a été abandonnée au profit de celles qui commencent par transformer la matrice A en une matrice tridiagonale T qui lui est orthogonalement semblable. Cette réduction se fait en un nombre déterminé d'étapes (n-2 transformations) ; la partie itérative de l'algorithme est alors appliquée à la matrice T, ce qui diminue notablement l'ordre des calculs. Ainsi, les bibliothèques scientifiques ont-elles leurs programmes, recherchant les valeurs et vecteurs propres bâtis sur ces méthodes. Les algorithmes ont été définis par Wilkinson et Reinsch[WiRe 71] et implémentés dans la bibliothèque EISPACK[Ei 76] avant d'être repris par les autres bibliothèques. Dans la suite de l'exposé, la bibliothèque EISPACK sera donc notre référence pour les algorithmes séquentiels. En ce qui concerne le problème symétrique, il existe plusieurs chemins, chacun d'eux correspondant à ce que l'on recherche :

<u>1er cas :</u> on recherche tous les vecteurs propres (et toutes les valeurs propres)



2e cas : on ne recherche que quelques vecteurs propres



Les subroutines TRED1 et TRED2 réduisent la matrice $A \in \mathbb{R}^{nxn}$ en une matrice T tridiagonale par (n-2) transformations de Housolder (symétries orthogonales par rapport à des hyperplans). Elles ne se distinguent que par l'information qu'elles conservent de ces transformations. Dans TRED1, on ne retient que la suite des vecteurs qui définissent ces transformations ce qui permet ensuite de les appliquer aux vecteurs propres de T pour les transformer en vecteurs propres de A par TRBAK1. Par contre dans TRED2, ces transformations sont composées entre elles pour obtenir la matrice Q qui sert de paramètre d'entrée à la routine TQL2. La procédure TRED1 est deux fois moins coûteuse en opérations que TRED2.

Le calcul des valeurs et vecteurs propres de T peut se faire par deux types d'algorithmes :

- méthode QL (TQL1 ou TQL2)
- méthodes des bisections (BISECT) et des itérations inverses (TINVIT).

Si maintenant on étudie la situation sous l'angle du *parallélisme* on retrouve le même type de classification.

Dans ce nouveau contexte, les méthodes de Jacobi ont repris de l'importance car il existe des politiques de choix d'élément à annuler qui sont fortement parallèlisables [Sa 71]. Cependant, comme le nombre d'opérations reste d'ordre $0(n^3)$ pour chaque balayage, cette méthode ne reste compétitive que pour de petites matrices, par exemple d'ordre inférieur à 100 pour un ALLIANT FX/8.[BeSa86]. Cela explique pourquoi, dans cette étude, nous nous sommes limités aux méthodes qui permettent de parallèliser les méthodes QL et des bisections. Les paragraphes suivants traitent en détail leurs implémentations sur un calculateur de type OPSILA. Avant cette étude, celle de la tridiagonalisation est aussi menée. Pour être complet, il sera nécessaire dans une étude ultérieure, de traiter la parallélisation de la méthode des itérations inverses : celle-ci ne pose pas de difficultés, puisque les tâches permettant le calcul de chaque vecteur propre recherché sont indépendantes. Malheureusement, la situation n'est pas tout à fait aussi simple car dans le cas de valeurs propres voisines il faut aussi assurer l'orthogonalité des vecteurs propres correspondants[LoPS 86].

4 - TRED1 - VECT et TRED2 - VECT : Versions parallèles de TRED1 et TRED2

4.1. : Principes de l'algorithme

Nous reprenons ici les algorithmes TRED1 et TRED2 de la bibliothèque EISPACK [Ei 76].

Ces algorithmes permettent de réduire une matrice réelle symétrique A d'ordre n à une matrice semblable tridiagonale symétrique T, par une suite de (n-2) transformations orthogonales de Householder.

L'algorithme TRED1 s'utilise lorsque seules les valeurs propres de la matrice A sont requises ou lorsque certaines valeurs propres et les vecteurs propres correspondants sont requis.

L'algorithme TRED2 s'utilise lorsque toutes les valeurs propres et tous les vecteurs propres de la matrice A sont requis.

La transformation i est de la forme :

$$A^{(i+1)} = P_i A^{(i)} P_i \qquad i = 1,2,\dots, n-2 \qquad A^{(1)} = A$$

où $P_i = I - u_i u_i^t / H_i$

$$H_i = u_i^t u_i/2$$

$$u_i^t = (a^{(i)}_{n-i+1,1} ; a^{(i)}_{n-i+1,2} ; \dots ; a^{(i)}_{n-i+1,n-i} \pm \sigma_i^{1/2} ; 0 ; \dots ; 0)$$

$$\sigma_i = \sum_{j=1}^{n-i} a^{(i)2}_{n-i+1,j}$$

La matrice P_i est construite de manière à annuler les termes $(a^{(i+1)}_{n-i+1,1}, \dots, a^{(i+1)}_{n-i+1,n-i-1})$. La transformation préserve la symétrie.

Si z est vecteur propre de la matrice tridiagonale $T = A^{(n-1)}$, alors $P_1 P_2 \dots P_{n-2} z$ est vecteur propre de la matrice A.

4.2. : Implémentation

Les algorithmes TRED1-VECT et TRED2-VECT (Annexe 2) sont purement vectoriels, de type SIMD.

Si on note

$q_i = A^{(i)} u_i/H_i - k_i u_i$

où

$k_i = u_i^t A_i u_i/2H_i^2$

la transformation i devient :

$A^{(i+1)} = A^{(i)} - u_i q_i^t - q_i u_i^t$

La matrice $A^{(i+1)}$ diffère de la matrice $A^{(i)}$ par les lignes et colonnes d'indices $1,2,\dots,n-i+1$.

L'algorithme TRED1 préserve les informations sur les matrices P_i : les vecteurs u_i sont stockés dans la partie triangulaire inférieure de A.

Dans l'algorithme TRED2, on calcule la matrice

$Q = P_1 P_2 \dots P_{n-2}$

Cette matrice se détermine de la manière suivante :

$Q_{n-2} := P_{n-2}$

$Q_i := Q_{i+1}\ P_i \qquad i = n-3, \dots, 1 \qquad Q_1 = Q.$

La matrice est de la forme :

$$\left(\begin{array}{c|c|c} Q_{11} & 0 & O \\ \hline 0 & 1 & O \\ \hline O & 0 & I_{i-1} \end{array}\right) \begin{array}{l} \}\ n-i \\ \}\ 1 \\ \}\ i-1 \end{array}$$

A l'étape i, il suffit de calculer le bloc de Q constitué de ses (n-i) premières lignes et colonnes.

On introduit dans l'algorithme TRED2-VECT le vecteur TRANSFORM : TRANSFORM(i) indique si la transformation

$$A^{(i+1)} = P_i\ A^{(i)}\ P_i$$

a eu lieu, ceci pour éviter une multiplication par l'identité dans le calcul de la matrice Q.

4.3. : Résultats des simulations

Les simulations sont faites sur une matrice d'ordre 32 (M_{32}) et une matrice d'ordre 64 (M_{64})

Pour chaque algorithme, un tableau récapitule

- la durée d'exécution t pour une configuration donnée (4,8,16 processeurs)
- l'accélération σ par rapport à une configuration à 4 processeurs.

Tous les tests sont effectués avec un nombre de bancs mémoire parallèle de données égal au nombre de processeurs.

4.3.1 : algorithme TRED1-VECT

Nombre de processeurs	TRED1-VECT
4	t= 0.207 σ = 1
8	t= 0.129 σ= 1.6
16	t= 0.097 σ= 2.1

t : unité 10^6 cycles

Tableau 4.1 : Temps OPSILA et accélérations par rapport à 4 processeurs, pour la matrice M_{32}

Nombre de processeurs	TRED1-VECT
4	t= 1.478 σ = 1
8	t= 0.808 σ = 1.8
16	t= 0.492 σ = 3.0

t : unité 10^6 cycles

Tableau 4.2 : Temps OPSILA et accélérations par rapport à 4 processeurs, pour la matrice M_{64}

4.3.2 : algorithme TRED2-VECT

Nombre de processeurs	TRED2-VECT
4	t= 0.394 σ= 1
8	t= 0.232 σ= 1.7
16	t= 0.164 σ= 2.4

t : unité 10^6 cycles

Tableau 4.3 : Temps OPSILA et accélérations par rapport à 4 processeurs, pour la matrice M_{32}

Nombre de processeurs	TRED2-VECT
4	t= 2.958 σ= 1
8	t= 1.573 σ= 1.9
16	t= 0.907 σ= 3.3

t : unité 10^6 cycles

Tableau 4.4 : Temps OPSILA et accélérations par par rapport à 4 processeurs, pour la matrice M_{64}

4.4. : Commentaire des résultats

Les algorithmes TRED1-VECT et TRED2-VECT sont purement vectoriels. L'accroissement du nombre de processeurs entraine donc un accroissement de l'accélération. Cet accroissement n'est cependant pas proportionnel à celui du nombre de processeurs. Ceci est dû au fait que ces algorithmes traitent des vecteurs de plus en plus courts, or l'efficacité des calculs diminue avec la longueur du vecteur parce que pour une tranche incomplète certains processeurs sont inactifs et aussi parce que le temps de préparation de l'instruction vectorielle devient prohibitif relativement au calcul (cf §7.3).

5 - TQL1 VECT et TQL2-VECT : Versions parallèles de TQL1 et TQL2

5.1. Principes de l'algorithme

L'algorithme étudié est l'algorithme TQL2 mis en oeuvre dans [Ei 76]. Il permet de déterminer toutes les valeurs propres et tous les vecteurs propres d'une matrice tridiagonale symétrique d'ordre n.

L'algorithme TQL1 est extrait de TQL2. Il ne calcule pas la matrice des vecteurs propres.

5.1.1 : calcul des valeurs propres

Soit T la matrice tridiagonale initiale. L'algorithme QL consiste à annuler successivement les éléments hors-diagonaux de T.

Une itération s est définie par :

$$\begin{cases} T_s = Q_s L_s \\ T_{s+1} = L_s Q_s \end{cases} \qquad s = 1, 2, \ldots \qquad T_1 = T$$

La matrice L_s est triangulaire inférieure.

La matrice Q_s est orthogonale. C'est le produit de (n-1) rotations :

$$Q_s = P_1 P_2 \ldots P_{n-1}$$

$$P_i = \begin{bmatrix} 1 & & & & & \\ & \ddots & & & & \\ & & 1 & & & \\ & & & c_i & -s_i & \\ & & & s_i & c_i & \\ & & & & & 1 \\ & & & & & & \ddots \\ & & & & & & & 1 \end{bmatrix} \begin{matrix} \\ \\ \\ \leftarrow i \\ \leftarrow i+1 \\ \\ \\ \\ \end{matrix} \qquad c_i^2 + s_i^2 = 1$$

En pratique, on travaille sur la matrice $(T_s - k_s I)$, k_s étant choisi de manière à accélérer la convergence.

L'itération s devient alors :

$$\begin{cases} T_s - k_s I = Q_s L_s \\ T_{s+1} = L_s Q_s + k_s I \end{cases} \qquad s = 1,2,\ldots \qquad T_1 = T$$

Soit T_s la matrice tridiagonale :

$$T_s = \left[e_k^{(s)} \quad d_k^{(s)} \quad e_{k+1}^{(s)} \right]_{k=1,n}$$

L'itération s s'exprime sous la forme : (E1)

$$p_n = d_n^{(s)} - k_s \qquad c_n = 1 \qquad s_n = 0$$

$$\left.\begin{aligned} r_i &= (p_i^2 + e_i^{(s)2})^{1/2} \\ c_{i-1} &= p_i / r_i \\ s_{i-1} &= e_i^{(s)} / r_i \\ p_{i-1} &= c_{i-1} (d_{i-1}^{(s)} - k_s) - s_{i-1} c_i e_i^{(s)} \\ d_i^{(s+1)} &= c_i p_i + s_{i-1} (c_{i-1} c_i e_i^{(s)} + s_{i-1} (d_{i-1}^{(s)} - k_s)) \\ e_{i+1}^{(s+1)} &= s_i r_i \end{aligned}\right] \quad i=n,n-1,\ldots,2$$

$$e_2^{(s+1)} = s_1 p_1$$

$$d_1^{(s+1)} = c_1 p_1$$

Si l'itération s consiste à annuler l'élément $e_r^{(s)}$, k_s est déterminé comme étant la valeur propre de la sous-matrice $\begin{bmatrix} d_{r-1}^{(s)} & e_r^{(s)} \\ e_r^{(s)} & d_r^{(s)} \end{bmatrix}$ la plus proche de $d_{r-1}^{(s)}$.

Avant chaque itération, la suite des éléments hors-diagonaux non encore transformés est examinée. Si certains de ces éléments sont négligeables, la matrice T_s éclate en une somme directe de matrices tridiagonales, dont les valeurs propres sont calculées indépendamment.

5.1.2. : calcul des vecteurs propres

La matrice Q des vecteurs propres de la matrice tridiagonale T est le produit de toutes les matrices de transformations Q_s.

$$Q = Q_1 \; Q_2 \; \dots \; Q_{iter}$$

Dans notre étude, la matrice T est le résultat d'une tridiagonalisation par la méthode de Householder. La matrice Q d'entrée dans TQL2 est calculée dans la phase de tridiagonalisation. Elle contient le produit des matrices de transformations de Householder.

Chaque itération de l'algorithme QL produit sur Q la transformation :

$$Q := Q \;\; Q_s$$

La matrice Q finale est donc la matrice des vecteurs propres de la matrice pleine réelle symétrique à diagonaliser.

5.2. : Implémentation

Nous avons mis en oeuvre deux versions de l'algorithme TQL1-VECT et deux versions de l'algorithme TQL2-VECT. Nous ne parlons ici que de TQL2-VECT, l'algorithme TQL1-VECT s'en déduisant par la suppression du calcul de la matrice des vecteurs propres.

Une première version, TQL2-VECT.Seq reprend exactement l'algorithme TQL2 de EISPACK. Les seules modifications apportées consistent à exploiter l'aspect vectoriel inhérent à l'algorithme, notamment dans le calcul des vecteurs propres, et ne changent pas le fond de l'algorithme.

la deuxième version TQL2-VECT.Simd est issue d'une étude de Sameh et Kuck [SaKu 77], qui permet de déterminer simultanément les éléments $(c_i)_{i=1,n-1}$ et $(s_i)_{i=1,n-1}$ des matrices de rotation P_i et par suite, les éléments $(d_i^{(s+1)})_{i=1,n}$ et $(e_i^{(s+1)})_{i=2,n}$.

5.2.1 calcul des valeurs propres par l'algorithme TQL2-VECT.simd

En transformant les équations décrites en (E1), on se ramène à résoudre deux récurrences linéaires :

$$(R1) \quad \begin{cases} y_i = d'^{(s)}_i \; y_{i+1} - e^{(s)2}_{i+1} \; y_{i+2} & i=2,\dots,n-1 \\ y_n = d'^{(s)}_n \\ y_{n+1} = 1 \end{cases}$$

$d'^{(s)}_i$ désigne ici la quantité $d^{(s)}_i - k_s$

$$(R2) \quad \begin{cases} x_i = e^{(s)2}_i \; x_{i+1} + y_i^2 & i=2,\dots,n \\ x_{n+1} = 1 \end{cases}$$

Les quantités $(r_i)_{i=2,n}$, $(c_i)_{i=,n-1}$ et $(s_i)_{i=1,n-1}$ s'expriment en fonction des solutions des systèmes (R1) et (R2) :

$$\begin{cases} r_i^2 = x_i \,/\, x_{i+1} & i = 2,n \\ c_i = y_{i+1} \,/\, \sqrt{x_{i+1}} & i = 1,n-1 \\ s_i = e^{(s)}_{i+1} / |r_{i+1}| & i = 1,n-1 \end{cases}$$

Les éléments de la matrice tridiagonale T_{s+1} sont donnés par les relations :

$$\begin{cases} d^{(s+1)}_i = y_i \; y_{i+1} \; (r_i^2 + e^{(s)2}_{i-1}) / \; x_i + s^2_{i-1} \; d'^{(s)}_{i-1} & i = 2,n \\ e^{(s+1)}_{i+1} = s_i \; r_i & i = 2,n-1 \end{cases}$$

$$d_1^{(s+1)} = c_1 \, k_1$$
$$e_2^{(s+1)} = s_1 \, k_1$$

où $k_1 = (y_2 \, d'^{(s)}_1 - y_3 \, e_2^{(s)2}) / \sqrt{x_2}$

Une itération de l'algorithme QL parallèle consiste donc à résoudre les systèmes (R1) et (R2). On en déduit alors la matrice T_{s+1} par calcul vectoriel.

Les systèmes (R1) et (R2) se résolvent vectoriellement par la technique du "Recursive doubling" [CheKu 75] , [St 73] , consistant à calculer pas à pas les solutions d'une récurrence linéaire d'ordre n.

Soit à résoudre la récurrence linéaire du 1er ordre :

$$(S1) \qquad \begin{cases} x_n = b_n \\ x_i = a_i \, x_{i+1} + b_i \qquad i = 1, n-1 \end{cases}$$

On note E_1 la matrice définie par :

$$(E_1)_{ij} = \begin{cases} a_i \text{ si } j = i+1 & i=1,n \; ; \; j=1,n. \\ 0 \text{ ailleurs} \end{cases}$$

la récurrence (S1) s'exprime sous la forme :

$$(I - E_1) \, X = B_1 \qquad X = (x_1,\dots,x_n)^t \qquad B_1 = (b_1,\dots,b_n)^t$$

En multipliant à gauche les deux membres de l'égalité, par $(I + E_1)$, il vient :

$$(I - E_1^2) \, X = B_1 + E_1 \, B_1$$

que l'on note :

$$(I - E_2) \, X = B_2.$$

Si on note E_{2^r} la matrice définie par :

$$(E_{2^r})_{ij} = (E^2_{2^{r-1}})_{ij} = \begin{cases} \prod_{k=i}^{i+2^r-1} a_k & \text{si } j=i+2^r \\ 0 & \text{ailleurs} \end{cases} \qquad i=1,n \; ; \; j=1,n.$$

et B_{2r} la matrice définie par :

$B_{2r} = B_{2r-1} + E_{2r-1} B_{2r-1}$

on a à l'étape r la relation :

$(I - E_{2r}) X = B_{2r}$

La multiplication à gauche de chacun des deux membres de l'égalité par $(I + E_{2r})$ donne :

$$(I - E_{2r+1})X = B_{2r+1}$$

Lorsque $r = \lceil \log_2 n \rceil$, la matrice E_{2r} est nulle, et le vecteur B_{2r} contient les solutions du système S1.

Si on note A le vecteur $(a_1, ..., a_{n-1}, 1)^t$

B le vecteur $(b_1, ..., b_n)^t$

l'algorithme de résolution du système (S1) est donné par :

```
pour r := 1  jusqu_a ⌈log₂n⌉
  boucle
    pour_tout i dans [1, n-2^(r-1)] :
        B(i) := B(i) + A(i) * B(i+2^(r-1))
        A(i) := A(i) * A(i+2^(r-1))
    fin_pour_tout
  fin_boucle .
```

Une récurrence linéaire du second ordre, telle que (R2), se résoud de la même manière qu'une récurrence linéaire du premier ordre, à ceci près que les opérations sur les scalaires se transforment en opérations matricielles :

En effet, soit à résoudre le système :

$$(S2) \quad \begin{cases} y_{n+1} = 0 \\ y_n = c_n \\ y_i = a_i\, y_{i+1} + b_i\, y_{i+2} + c_i & i = 1, n-1 \end{cases}$$

Ce système se met sous la forme :

$$\begin{cases} Y_n = C_n \\ Y_i = A_i\, Y_{i+1} + C_i \end{cases}$$

où $Y_i = {}^t(y_i, y_{i+1})$, $C_i = {}^t(c_i, 0)$

$$A_i = \begin{bmatrix} a_i & b_i \\ 1 & 0 \end{bmatrix}$$

On est donc ramené à une récurrence linéaire d'ordre un.

5.2.2 : calcul des vecteurs propres

Une itération de l'algorithme TQL2-VECT effectue sur la matrice Q des vecteurs propres la transformation :

$$Q := Q\,Q_s$$
$$= Q\,P_1\,P_2 \dots P_{n-1}$$

La multiplication à droite d'une matrice par une matrice de rotation P_i consiste à combiner linéairement les colonnes i et i+1 de cette matrice. Cette opération est purement vectorielle.

5.2.3 : schéma général de l'algorithme TQL2-VECT

Les éléments diagonaux et hors-diagonaux de la matrice T sont stockés respectivement dans les vecteurs D et E.

tant que $E \neq 0$ faire

rechercher le premier bloc tridiagonal $T_{rt} = [e_k\ d_k\ e_{k+1}]_{k=r,t}$ de la matrice T

calculer le shift k_s ;

tant que $e_{r+1} \neq 0$ faire

Option Séquentielle	*Option Vectorielle*
	résoudre (R1) sur T_{rt}
effectuer (E1) sur T_{rt}	résoudre (R2) sur T_{rt}
	calculer D
	calculer E

calculer la matrice des vecteurs propres

fait

fait

5.3. : Résultats des simulations

Les simulations sont effectuées sur les matrices W^{+}_{21}, L_{32} et L_{64} décrites en annexe 5.

Les calculs sont faits en simple précision (flottants 36 bits). Pour chaque algorithme (TQL1-VECT et TQL2-VECT) un tableau récapitule :

- la durée d'exécution t pour une configuration donnée (4,8,16 processeurs)
- l'accélération σ par rapport à une configuration à 4 processeurs.

Dans les configurations choisies, le nombre de bancs mémoire parallèle de données est égal au nombre de processeurs.

5.3.1 : algorithme TQL1-VECT

Nombre de processeurs	TQL1-VECT.Seq	TQL1-VECT.Simd
4	t= 0.147 σ= 1	t= 0.193 σ= 1
8	t= 0.146 σ≃ 1	t= 0.135 σ=1.4
16	t= 0.145 σ≃ 1	t= 0.109 σ=1.8

t : unité 10^6 cycles

Tableau 5.1 : Temps OPSILA et accélérations par rapport à 4 processeurs pour la matrice W^+_{21}

Nombre de processeurs	TQL1-VECT.Seq	TQL1-VECT.Simd
4	t= 0.358 σ= 1	t= 0.475 σ= 1
8	t= 0.355 σ≃ 1	t= 0.296 σ= 1.6
16	t= 0.354 σ≃ 1	t= 0.214 σ= 2.2

t : unité 10^6 cycles

Tableau 5.2 : Temps OPSILA et accélérations par rapport à 4 processeurs pour la matrice L_{32}

Nombre de processeurs	TQL1-VECT.Seq	TQL1-VECT.Simd
4	t= 1.404 σ= 1	t= 2.213 σ= 1
8	t= 1.393 $\sigma \simeq 1$	t= 1.218 σ= 1.8
16	t= 1.388 $\sigma \simeq 1$	t= 0.738 σ= 3.0

t :unité 10^6 cy

Tableau 5.3 : Temps OPSILA et accélérations par rapport à 4 processeurs pour la matrice L_{64}

5.3.2 : algorithme TQL2-VECT

Nombre de processeurs	TQL2-VECT.Seq	TQL2-VECT.Simd
4	t= 0.381 σ= 1	t= 0.451 σ= 1
8	t= 0.274 σ= 1.4	t= 0.269 σ= 1.7
16	t= 0.229 σ= 1.7	t= 0.191 σ= 2.4

t:unité 10^6 cycl

Tableau 5.4 : Temps OPSILA et accélérations par rapport à 4 processeurs pour la matrice W_{21}^{+}

Nombre de processeurs	TQL2-VECT.Seq	TQL2-VECT.Simd
4	t= 1.190 σ= 1	t= 1.328 σ= 1
8	t= 0.793 σ= 1.5	t= 0.728 σ= 1.8
16	t= 0.595 σ= 2.0	t= 0.433 σ= 3.1

t:unité 10^6 cycles

Tableau 5.5 : Temps OPSILA et accélérations par rapport à 4 processeurs pour la matrice L_{32}

Nombre de processeurs	TQL2-VECT.Seq	TQL2-VECT.Simd
4	t= 8.013 σ= 1	t= 9.015 σ= 1
8	t= 4.788 σ= 1.7	t= 4.622 σ= 1.9
16	t= 3.176 σ= 2.5	t= 2.446 σ= 3.7

t:unité 10^6 cycle

Tableau 5.6 : Temps OPSILA et accélérations par rapport à 4 processeurs pour la matrice L_{64}

5.4. : Commentaire des résultats

* L'algorithme TQL1-VECT.Seq ne fait pratiquement pas intervenir de calcul vectoriel. Il est donc normal que l'accroissement du nombre de processeurs n'influe pas sur les durées d'exécution.

* L'algorithme TQL2-VECT.Seq bénéficie de l'accroissement du nombre de processeurs dans le calcul de la matrice des vecteurs propres, ce calcul étant purement vectoriel.

* L'algorithme TQL1-VECT.Simd bénéficie de l'accroissement du nombre de processeurs, mais dans une faible mesure. Les raisons en sont les suivantes :

- Au fur et à mesure que l'on avance dans l'algorithme, les blocs tridiagonaux traités, et donc les vecteurs qui en sont issus, sont de plus en plus courts.
- Pour une itération donnée, la boucle du "Recursive doubling" (§5.2.1) travaille sur des vecteurs dont la longueur diminue de moitié à chaque passage.

Le travail sur des vecteurs courts est indépendant du nombre de processeurs.

* Pour l'algorithme TQL2-VECT.Simd, les mêmes remarques que pour TQL1-VECT.Simd sont à faire. Cependant, les accélérations observées sont supérieures. Ceci est dû au calcul de la matrice des vecteurs propres qui est purement vectoriel.

La vectorisation d'une itération de l'algorithme QL par la technique du Recursive Doubling est coûteuse en temps de calcul car elle introduit un surcroît de calcul. Lorsque le nombre de processeurs est petit, il est donc préférable d'utiliser la version utilisant des itérations séquentielles. Nos tests montrent que la version vectorielle devient rentable à partir d'une configuration à 16 processeurs.

6 - TREPS : Version parallèle de BISECT

6.1. : Principes de l'algorithme

On considère ici une matrice tridiagonale et symétrique $T = [e_k \; d_k \; e_{k+1}] \in \mathbb{R}^{nxn}$ et un intervalle $I = [a,b]$.

Pour localiser les valeurs propres de T qui appartiennent à I, on utilise une propriété que possède la suite des mineurs principaux de T : le nombre de valeurs propres de T qui sont inférieures à une valeur λ donnée est égal au nombre de changements de signes dans la suite $\{p_k(\lambda) = \det (T_k - \lambda I_k)\}$; cette suite est appelée *suite de Sturm*. La forme particulière de la matrice T entraine l'existence d'une relation de récurrence qui permet le calcul de cette suite :

$$\begin{cases} p_0 (\lambda) = 1 \\ p_1 (\lambda) = d_1 - \lambda \\ p_k (\lambda) = (d_k - \lambda) \, p_{k-1} (\lambda) - e_k^2 \, p_{k-2} (\lambda), \; k=2,\dots,n \end{cases} \qquad (6\text{-}1)$$

Malheureusement, dans certains cas, ce calcul entraine des dépassements de l'intervalle des nombres représentables en machine (under- ou overflow). Pour y remédier on préfère "normaliser" cette suite en considérant la nouvelle suite :

$$q_k (\lambda) = p_k (\lambda) \, / \, p_{k-1} (\lambda) \; , \; k = 1,\dots,n$$

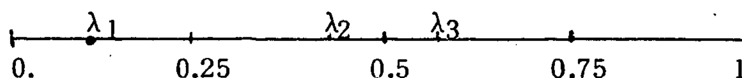
La relation de récurrence (6-1) devient alors :

$$\begin{cases} q_1 (\lambda) = d_1 - \lambda \\ q_k (\lambda) = d_k - \lambda - e_k^2 \; / \; q_{k-1} (\lambda) \; , \; k = 2,\dots,n \end{cases} \qquad (6\text{-}2)$$

et on compte alors le nombre de termes négatifs dans la suite $\{q_k(\lambda)\}$.

La *partition de l'intervalle* [a,b] peut donc se faire en calculant itérativement les suites de Sturm en des points de plus en plus proches des valeurs propres cherchées. La méthode la plus simple consiste à opérer par bisections et à associer à chaque valeur propre un sous-intervalle de [a,b] auquel elle appartient. L'amplitude de cet intervalle est réduite de moitié à chaque bisection. *Par exemple*, supposons que

l'on recherche les valeurs propres dans [0,1] et que, par un calcul des suites de Sturm en 0 et en 1, on sache qu'il y en a trois dans cet intervalle réparties de la manière suivante :



Les encadrements pour chaque valeur propre seront alors :

	au départ	après bisect en $\lambda = 0.5$	après bisect en $\lambda = 0.25$	
λ_1	[0. , 1.]	[0. , 0.5]	[0. , 0.25]	
λ_2	[0. , 1.]	[0. , 0.5]	[0.25 , 0.5]	. . . .
λ_3	[0. , 1.]	[0.5 , 1.]	[0.5 , 1.]	

Cette politique est celle qui est utilisée dans la subroutine BISECT.

Avant d'aborder l'introduction du parallèlisme dans cet algorithme, nous allons l'étudier plus précisément. On suppose maintenant que la politique de choix des intervalles à partager correspond à *isoler* dans un premier temps toutes les valeurs propres demandées, c'est-à-dire à déterminer pour chacune d'elles un intervalle dans lequel elle soit seule valeur propre de T. La deuxième étape permet alors d'*extraire* de chacun de ces intervalles la valeur propre qui lui appartient.

L'*extraction* d'une valeur propre d'un intervalle donné consiste à calculer le seul zéro de l'équation

$$p_n(\lambda) = 0$$

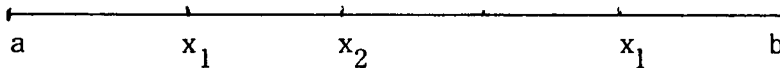
qui appartienne à cet intervalle. Cela peut se faire, nous l'avons vu, par bisections mais cette méthode est d'ordre 1. (Une méthode itérative qui définit une suite $\{x_m\}$ convergeant vers x est d'ordre r lorsque $|x_{m+1} - x| = 0\ |x_m - x|^r$). On pourrait utiliser la méthode de Newton qui converge quadratiquement mais elle nécessite à chaque étape le calcul de la fonction et de sa dérivée. On lui préfère la méthode de la sécante qui est d'ordre plus faible $((1 + \sqrt{5})\ /2)$ mais ne requiert que l'évaluation de la fonction à chaque étape. Il existe une procédure appelée ZEROIN [FoMM 77] qui combine habilement la méthode de la sécante et celle de la bisection ; elle tente même à chaque pas une extrapolation quadratique. C'est à cet algorithme que l'on se réfèrera lorsque,

dans la suite, on citera la méthode de la sécante. Malgré sa relative lenteur, l'algorithme des bisections tire son avantage du fait de pouvoir utiliser la suite $\{q_k(\lambda)\}$ dont le calcul est numériquement plus robuste que celui de la suite $\{p_k(\lambda)\}$. Pour pallier à cette difficulté, lorsque l'on utilise la méthode de la sécante, on calcule la suite $\{p_k(\lambda)\}$ à partir de la suite $\{q_k(\lambda)\}$ par :

$$\begin{cases} p_1(\lambda) = q_1(\lambda) = d_1 - \lambda \\ \left.\begin{array}{l} \text{calcul de } q_k(\lambda) \text{ par la formule (6-2)} \\ p_k = q_k \cdot p_{k-1} \end{array}\right\} \quad k = 2, \dots, n \end{cases} \tag{6-3}$$

Le nombre d'opérations en est évidemment accru.

Le *parallèlisme* sera obtenu de manière différente dans les deux étapes, mais dans chaque cas il se fera par l'évaluation simultanée de plusieurs suites de Sturm. Dans la deuxième phase, les tâches de calcul correspondant à extraire les valeurs propres sont évidemment indépendantes ; le niveau de parallèlisme de cette étape est donc fixé par le nombre de valeurs propres recherchées. Par contre, pour la phase de séparation on introduit la notion de multisection : appliquer une *multisection d'ordre* l sur un intervalle consiste à calculer la suite de Sturm $\{q_k(x)\}$ en l points régulièrement espacés dans l'intervalle :



Une bisection est donc une multisection d'ordre 1. En général l'ordre des multisections sera induit par le nombre de processeurs dont on dispose. On pourrait a priori envisager l'emploi des multisections dans la phase d'extraction, mais un simple calcul [LoPS 86] montre que pour extraire une valeur propre d'un intervalle la multisection d'ordre l a, par rapport à la bisection, une efficacité

$$E_l = [\log_2 (l+1)] / l$$

ce qui ne permet pas de choisir une valeur élevée pour l.
($E_4 \simeq 58$ % ; $E_8 \simeq 40$ % ; $E_{16} \simeq 26$ %) on réserve donc l'emploi des multisections à la phase de séparation.

A partir de ces principes un algorithme a donc été conçu : TREPS (TRidiagonal Eigenvalue Parallel Solver). Les grandes lignes de cet algorithme sont récapitulées dans la Table 6.1. Cet algorithme a deux versions (TREPS1 ou TREPS2) suivant que l'extraction est réalisée par bisections ou par ZEROIN. Examinons maintenant l'implémentation de ces algorithmes sur l'architecture ciblée.

<u>Faire</u>

tant qu'il reste des sous-matrices à étudier

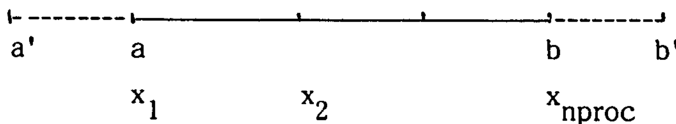
trouver la dernière ligne q de la sous-matrice courante.

($q = n$ ou $e_{q+1} = 0$)

calcul du domaine de Gershgorin et réduction éventuelle de l'intervalle de départ

procéder à la première multisection :

(celle-ci correspond à une multisection d'ordre nproc sur un intervalle fictif qui permet le calcul des suites de Sturm aux bornes du réel intervalle :



cette multisection permet de calculer le nombre et le rang des valeurs propres à rechercher pour la sous-matrice courante).

<u>co</u> séparation des valeurs propres <u>fco</u>

<u>faire</u>

tant qu'il reste des intervalles avec plusieurs valeurs propres

procéder à des multisections d'ordre nproc

<u>fait</u>

<u>co</u> extraction des valeurs propres <u>fco</u>

<u>faire</u>

tant qu'il reste des valeurs propres à extraire

prendre un intervalle et en extraire la valeur propre

<u>fait</u>

fusionner la liste triée des valeurs propres de la sous-matrice à la liste des valeurs propres des sous-matrices précédentes.

<u>fait</u>

<u>Tableau 6-1</u> : Plan général de l'algorithme TREPS

6.2. : Implémentation de TREPS en modes SIMD et SPMD

L'organisation générale de la *structure des données* est commune pour les différentes implémentations.

La matrice est stockée par l'intermédiaire de sa diagonale D et de sa sous diagonale E ; l'intervalle [bg, bd] est le domaine de recherche des valeurs propres.

Trois tableaux décrivent l'information relative aux sous-matrices :

LROW(k) : indice de la dernière ligne de la $k^{ème}$ sous-matrice

NEVSUB(k) : nombre de valeurs propres issues de cette sous-matrice dans l'intervalle donné

RANKF(k) : rang (par rapport au spectre de la sous-matrice) de la plus petite valeur propre à conserver.

A un moment donné, l'avancement du calcul des valeurs propres d'une sous-matrice sera caractérisé par l'état de la "réserve" qui est décrite dans le tableau 6-2.

	borne gauche	borne droite	rang de la plus petite v.p.	nombre de v.p.	↓ intervalles contenant plusieurs valeurs propres
grd 2 →					
top 1 →					
	borne gauche	borne droite	rang de la v.p.	indice de la sous-matrice	↑ intervalles ne contenant qu'une valeur propre

Tableau 6-2 : Réserve de travail

Pour la *séparation* des valeurs propres, chaque multisection permet le calcul de nproc suites de Sturm en parallèle. Ce calcul est intrinsèquement de type SIMD, mais il subsiste une difficulté dûe au traitement d'un cas particulier (très rare en pratique) destiné à éliminer une division par zéro. On traite ce cas par masquage grâce à l'algorithme suivant :

```
pour i := p + 1 jusqu_a q
  boucle
    si (il existe un élément U(i) nul)
              alors
                passage de la tranche en deux fois
                  - pour traiter les i tq U(i) ≠ 0 ;
                  - pour traiter les i tq U(i) = 0 ;
              sinon
                passage de la tranche en une fois ;
    fin_si ;
  fin_boucle ;
```

Une deuxième version de cette partie a été programmée en SPMD en appelant en parallèle une procédure STURM qui calcule la suite de Sturm au point affecté au processeur.

Au contraire de la séparation, l'étape d'*extraction* est intrinsèquement SPMD, puisqu'il s'agit d'exécuter une boucle du type :

```
pour_tout i dans [1, topl] :
   EXTRAIRE(i)
fin_pour_tout ;
```

où le chemin d'exécution dans EXTRAIRE est différent pour chaque valeur de i. Nous avons vu qu'il existe deux algorithmes pour cette procédure EXTRAIRE suivant qu'elle utilise la méthode des bisections ou la méthode de la sécante. Pour que l'implémentation de la seconde soit numériquement fiable, on utilise la formule (6-3) pour le calcul des

suites de Sturm et on revient à l'algorithme des bisections (avec utilisation de la formule (6-2)) si la valeur de p_n (λ) devient trop petite (inférieure à 10^{-40}).

Une dernière formulation de l'extraction a tenté une écriture de la méthode des bisections en SIMD, grâce à l'algorithme suivant :

prendre nproc intervalles à réduire ;
boucle
tant qu'il reste des intervalles non réduits ;
procéder à une bisection sur ces intervalles (en parallèle) ;
remplacer les intervalles réduits par de nouveaux à réduire ;
fin_boucle ;

Dans cette formulation les bisections exécutées en parallèle peuvent être vectorisées en traitant les tests par masquage.

En conclusion, nous avons obtenu les implémentations décrites dans le tableau 6-3.

	extraction par bisections - SIMD -	extraction par bisections - SPMD -	extraction par sécantes - SPMD -
séparation SIMD	TREPS1.ii	TREPS1.ip	TREPS2.ip
séparation SPMD	———	———	TREPS2.pp

Tableau 6-3 : Différentes implémentations de TREPS

6.3. : Résultats des simulations

Pour tester ces programmes, on a considéré *deux matrices* de type différents qui sont décrites en annexe 5.

- La matrice L_n qui est un bon représentant de matrice à valeurs propres bien réparties.
- La matrice W_{21}^{+} d'ordre 21 qui a des couples de valeurs propres si voisines qu'on ne peut les séparer ; elles sont considérées comme numériquement confondues et leur calcul est complètement assuré par les multisections de la phase de séparation.

L'*ordre* de ces matrices n'a pu être choisi très grand car le temps d'exécution du simulateur est très élevé.
La *précision de calcul* des valeurs propres est donnée par une quantité $K\varepsilon$ où K est une constante qui dépend de la norme de la matrice et ε est le paramètre de précision. On a choisi $\varepsilon = 10^{-7}$.

Les calculs on été effectués en simple précision (36 bits).

Pour chaque programme, on a reporté dans les tableaux 6-4 et 6-5 le temps OPSILA d'exécution ainsi que l'accélération par rapport à 4 processeurs, et cela pour les matrices L_{64} et W_{21}^{+}.

	TREPS1.ii	TREPS1.ip	TREPS2.ip	TREPS2.pp
4 processeurs	t= 4.232 σ= 1.0	t= 2.070 σ= 1.0	t= 1.073 σ= 1.0	t= 3.117 σ = 1.0
8 processeurs	t= 2.467 σ= 1.7	t= 1.128 σ= 1.8	t= 0.629 σ= 1.7	t= 1.290 σ= 2.4
16 processeurs	t= 1.748 σ= 2.4	t= 0.784 σ= 2.6	t= 0.555 σ= 2.1	t= 1.348 σ= 2.3

t: temps
(unité:10^6 cycle
σ:accélération

Tableau 6-4 : Temps OPSILA et accélérations par rapport à 4 processeurs, pour la matrice L_{64}

	TREPS1.ii	TREPS1.ip	TREPS2.ip	TREPS2.pp
4 processeurs	t= 0.565 σ= 1.0	t= 0.430 σ= 1.0	t= 0.367 σ= 1.0	t= 3.291 σ= 1.0
8 processeurs	t= 0.404 σ= 1.4	t= 0.316 σ= 1.4	t= 0.291 σ= 1.3	t= 1.884 σ= 1.7
16 processeurs	t= 0.366 σ= 1.5	t= 0.290 σ= 1.5	t= 0.273 σ= 1.3	t= 1.198 σ= 2.7

t:temps
(unité:10^6 cycles)
σ:accélération

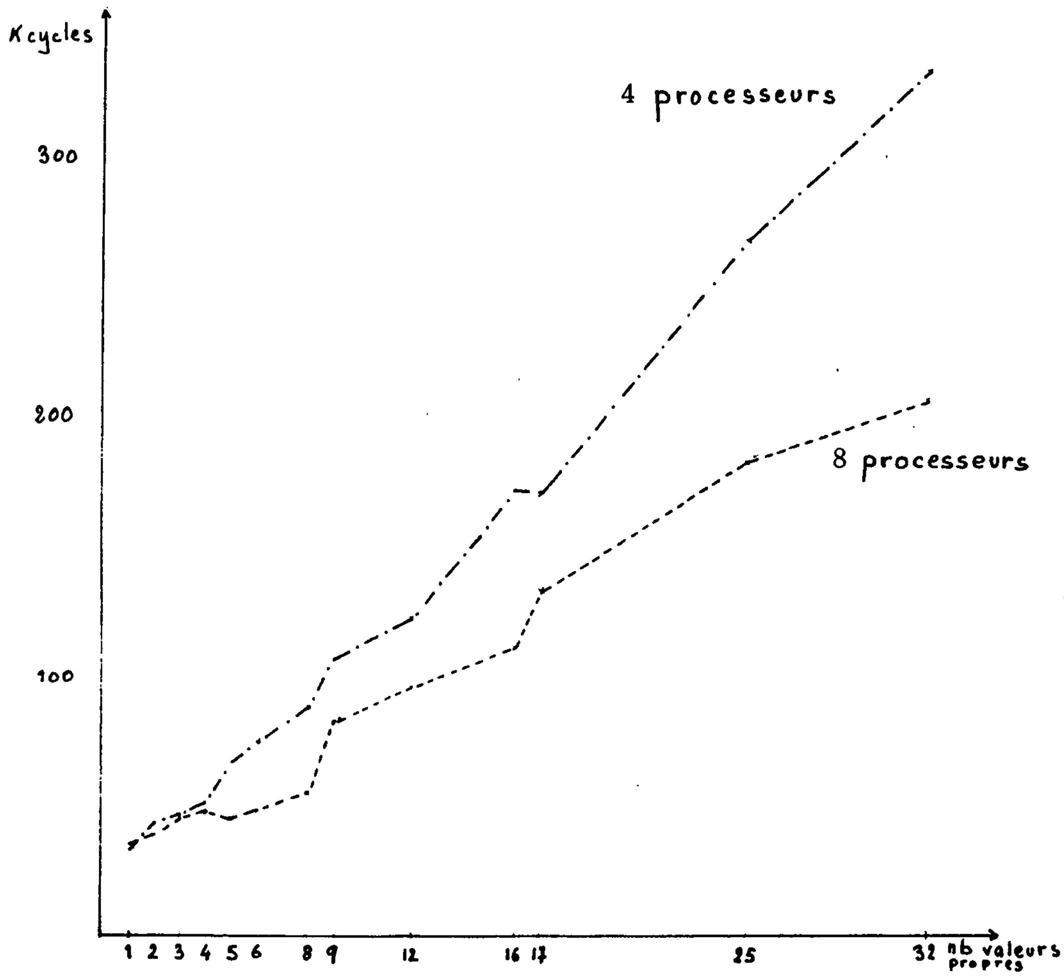
Tableau 6-5 : Temps OPSILA et accélérations par rapport à 4 processeurs, pour la matrice W_{21}^{+}

6.4. : Commentaires des résultats

Il ressort de ces essais que :

- L'algorithme de la sécante est plus rapide que celui des bisections pour l'extraction. (Comparaison TREPS1.ip et TREPS2.ip) Cette accélération est plus sensible pour la matrice L_{64} que pour W_{21}^{+} puisque pour cette dernière il n'y a que quelques valeurs propres qui sont effectivement extraites par la deuxième étape.
- L'implémentation SIMD de la multisection est plus efficace que l'implémentation SPMD (comparaison TREPS2.ip et TREPS2.pp). Cette différence de comportement est particulièrement visible sur la matrice W_{21}^{+} pour laquelle la majeure partie des calculs est concentrée dans l'étape de séparation.
- L'implémentation SPMD de la phase d'extraction est plus efficace que son implémentation SIMD (comparaison TREPS1.ii et TREPS1.ip). La différence de comportement est, au contraire du cas précédent, plus visible sur la matrice L_{64} pour la même raison.

En conclusion, la supériorité du programme TREPS2.ip étant nette, on tente d'apprécier son niveau de parallèlisme en fonction du nombre de valeurs propres à trouver. A cause du nombre d'essais à réaliser, on a effectué les mesures sur la matrice L_{32}. On a mesuré l'accélération entre 4 et 8 proceseurs estimant que 16 était un nombre trop élevé de processeurs par rapport à la taille de la matrice. Les résultats sont reportés dans la figure 6-6. On y voit nettement les sauts qui correspondent au passage à une tranche de plus, spécialement pour 8 processeurs. L'accélération entre 4 et 8 processeurs dépasse 1.5 lorsque le nombre des valeurs propres cherchées est suffisant. On ne peut qu'espérer une meilleure accélération pour de plus grandes matrices.



(Matrice L_{32})

Figure 6 - 6 : Temps d'exécution en fonction du nombre de valeurs propres cherchées

7 - Discussion

7.1. : Cas où seules les valeurs propres sont requises

Lorsque toutes les valeurs propres sont recherchées, le chemin recommandé est a priori :

TRED1 → TQL1_VECT (version TQL1-VECT-Simd).

Dans le cas contraire, on préfère le chemin

TRED1 → TREPS (version TREPS2.ip).

Cette distinction peut paraitre formelle. Suivant le type de la matrice manipulée, cette différence peut s'estomper : en effet, d'après les simulations sur la matrice L_{64}, TREPS est plus rapide que TQL1_VECT, cet avantage allant décroissant au fur et à mesure que le nombre de processeurs augmente.

Par contre, on observe un comportement inverse sur la matrice W_{21}^{+}. Cette situation peut s'expliquer par l'effet de la distribution des valeurs propres sur l'algorithme TREPS.

Pour une matrice à valeurs propres bien séparées(L_{64}), la majeure partie du travail de TREPS est produite dans l'étape d'extraction et cette étape présente un bon parallèlisme.

Lorsque la matrice présente des valeurs propres très voisines (numériquement confondues comme dans W_{21}^{+}), TREPS fournit le plus gros de son travail dans l'étape de séparation, qui dans ce cas se substitue à l'étape d'extraction. Cette situation entraine une dégradation dans l'efficacité (voir § 6-1).

L'algorithme QL présente un comportement plus régulier et est moins sensible à la distribution des valeurs propres.

D'autre part, on observe que TQL1_VECT maintient son efficacité au fur et à mesure que le nombre de processeurs augmente (voir § 5-4)

On peut qualitativement définir des domaines de mise en oeuvre des algorithmes TQL1_VECT et TREPS qui sont suggérés dans la figure 7-1.

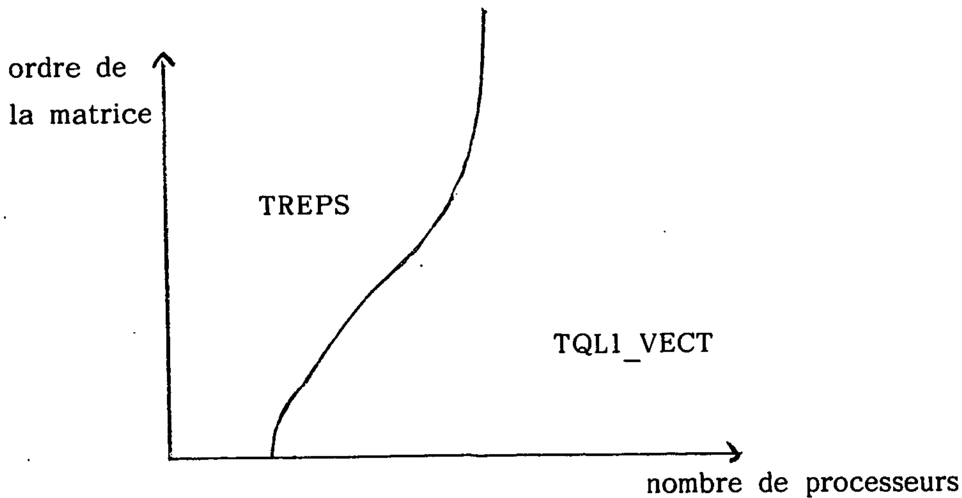


Figure 7-1 : Domaines de mise en oeuvre des algorithmes TREPS et TQL1_VECT

7.2. Cas où des vecteurs propres sont requis

Lorsque tous les vecteurs propres sont requis, le chemin recommandé est :

TRED2 → TQL2_VECT

Dans le cas contraire, on préfère le chemin

TRED1 → TREPS → TINVIT → TRBAK1

L'algorithme TINVIT n'a pas été implémenté en HELLENA. Il n'est donc pas possible de connaitre la proportion de vecteurs propres à partir de laquelle l'algorithme QL devient plus intéressant que celui de multisection. (Les auteurs de EISPACK situent à 25% cette proportion).

A priori, la supériorité de TQL2_VECT semble assurée dans la plupart des cas.

7.3. : Deux situations entrainant des dégradations

Nous exposons ici deux situations entrainant une dégradation de l'efficacité :

* L'exécution d'une instruction vectorielle (voir §2.1.1.) nécessite un travail préalable de l'U.G. :

- calcul des descripteurs des opérandes de l'instruction (type, adresse, raison interne, raison externe)
- calcul du descripteur de l'instruction (adresses de début et fin dans la mémoire de l'UCV, nombre d'itérations des boucles externe et interne).

Le temps de préparation d'une instruction vectorielle peut être démesuré par rapport à la durée d'exécution de cette instruction, lorsque celle-ci porte sur des vecteurs courts ou comporte peu d'opérations.

Par exemple, l'addition vectorielle :

```
pour_tout i dans [1,n]
  A(i) := B(i) + C(i) ;
fin_pour_tout ;
```

nécessite un temps de préparation par l'UG de 28 cycles, alors que l'instruction sur une tranche prend 13 cycles. Pour que l'UG ne prenne pas de retard sur l'UCV, il faut donc que la longueur du vecteur soit supérieure à deux tranches.

* Lorsqu'une instruction doit être exécutée par l'UG et que l'un de ses opérandes est un élément de tableau, il est nécessaire de ramener cet élément dans l'UG. Pour cela, il faudra préparer l'instruction vectorielle permettant de le faire. L'ensemble de la préparation et de la lecture en mémoire vectorielle représente 17 cycles. A titre indicatif, une addition flottante de scalaires ne coûte que 4 cycles.

7.4. : Pratique du SPMD

Dans ce paragraphe, nous tentons de regrouper quelques réflexions après l'utilisation du mode SPMD.

Ce mode est efficace lorsque la *granularité des tâches* est suffisante. Cela a été vérifié grâce aux différentes versions de TREPS. Par exemple dans la version SPMD de la procédure MUSECT, chaque tâche ne comporte que le calcul d'une suite de Sturm. Par contre dans la version SPMD de l'extraction chaque tâche comporte l'extraction de plusieurs valeurs propres, chacune d'elles faisant intervenir plusieurs calculs de suites de Sturm (spécialement pour l'extraction par bisections).

Pour utiliser le mode SPMD, il a fallu préparer les données par :
- une diffusion à tous les bancs des données à partager en lecture
- une identification de la réunion de tous les représentants d'un vecteur SPMD avec un vecteur SIMD, pour permettre la répartition des données locales à la tâche.

Le problème de la répartition d'une matrice et non plus d'un vecteur n'a pas été abordé dans cette étude. Il est important car il correspond à des algorithmes parallèles comme celui de la multiplication d'une matrice creuse par un vecteur. Il demandera à être étudié avec soin.

Enfin, une discussion soulève régulièrement la question de la largeur de la déclaration du SPMD. Actuellement cette largeur est fixée par le nombre commun de bancs et de processeurs. On pourrait imaginer qu'au niveau d'HELLENA, une largeur quelconque pourrait être définie. Pour le présent, nous n'y sommes pas favorables car l'usage du mode SPMD nécessite une connaissance de l'architecture de la machine et la généralisation à une largeur quelconque aurait tendance à éloigner l'usager de la situation réelle.

CONCLUSION

Nous avons vu que chacun des algorithmes décrits (TQL_VECT et TREPS) correspond à un domaine d'application propre. Dans tous les cas, leur parallèlisme est satisfaisant.

Avec le langage HELLENA on exprime aisément l'aspect vectoriel des algorithmes.

Le SPMD a permis d'obtenir une bonne efficacité dans les exécutions de programme lorsqu'il a pu être employé. Son usage est assez difficile car il demande un soin particulier dans le rangement des données, mais il est bien adapté pour exécuter des tâches totalement indépendantes, situation relativement courante en analyse numérique.

REFERENCES

[Au 85] M. AUGUIN - *Etude et réalisation de structures de calculs parallèles - Application aux méthodes numériques.* Thèse - Université de Nice - 3 Mai 1985 -

[BeSa 86] M. BERRY, A. SAMEH - *Multiprocessor Jacobi Algorithms for Dense Symmetric Eigenvalue and Singular Value Decompositions.* ICPP, August 19-22, 1986 - pp 433-440 -

[CheKu 75] S.C. CHEN, D. KUCK - *Time and parallel processor bounds for linear recurrence systems.* IEEE Trans. Computer C24-7, pp 701-717, 1975 -

[Ei 76] EISPACK GUIDE - *Matrix Eingensystem routines.* Lecture notes in Computer Science n°6 - Springer Verlag - 1976 -

[FoMM77] G.E. FORSYTHE, M.A. MALCOM and C.B. MOLER - *Computer Methods for Mathematical Computations.* Prentice Hall, 1977 -

[Je 86] Y. JEGOU - *HELLENA.* Rapport technique IRISA - 1986 -

[LoPS 86] S.S. LO, B. PHILIPPE, A. SAMEH - *A Multiprocessor Algorithm for the symmetric Tridiagonal Eigenvalue Problem.* SIAM J. of Sc. and Sta. Computing - Vol 8 - n°2 - Mars 1987 -

[Sa 71] A. SAMEH - *On Jacobi and Jacobi like algorithms for a parallel Computer.* Math-Comp. - Vol 25 pp 20-24, 1971 -

[SaKu 77] A. SAMEH, D. KUCK - *A Parallel QR - Algorithm for Symmetric Tridiagonal Matrices.* IEEE Trans. Comput., Vol C 26, pp 147-153 - 1977 -

[St 73] H. STONE - *An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations.* JACM, Vol 20, pp 27-38 - 1973 -

[Wi 65] J.H. WILKINSON - *The Algebraic Eigenvalue Problem.* Oxford - 1965 -

[WiRe 71] J.H. WILKINSON - C. REINSCH - *Handbook for Automatic Computation.* Vol 2, Linear Algebra, Springer Verlag - 1971 -

ANNEXE 1

INSTRUCTIONS DE L'UNITE DE GESTION

DUREE	INSTRUCTION
1	* lecture immediate
2	* lecture memoire absolue
2	* lecture memoire basee
2	* lecture memoire indexee
2	* lecture memoire indirecte
2	* lecture memoire indirecte registre
2	* lecture memoire postincrementee
2	* lecture memoire predecrementee
1	* lecture registre
2	* recuperation de scalaire dans la file
2	* ecriture memoire absolue
2	* ecriture memoire basee
2	* ecriture memoire indexee
2	* ecriture memoire indirecte
2	* ecriture memoire indirecte registre
2	* ecriture memoire postincrementee
2	* ecriture memoire predecrementee
1	* ecriture registre
1	* sauvegarde dans registre
2	* iteration
0	* emission instruction vectorielle vers la file
0	* emission descripteur de vecteur vers la file
0	* emission operande scalaire vers la file
1	* addition entier
4	* addition flottant
1	* soustraction entier
4	* soustraction flottant
1	* multiplication entier
4	* multiplication flottant
22	* division entier
22	* division flottant
1	* > entier
4	* > flottant
1	* < entier
4	* < flottant
1	* >= entier
4	* >= flottant
1	* <= entier
4	* <= flottant
1	* = entier
4	* = flottant
1	* = booleen
1	* # entier
4	* # flottant
1	* # booleen
1	* minimum entier
4	* minimum flottant
1	* maximum entier
4	* maximum flottant
22	* modulo
22	* reste
1	* et
1	* ou
1	* ou exclusif

```
 1      * echange sommets de pile
 1      * moins unaire entier
 4      * moins unaire flottant
 1      * non
 1      * valeur absolue entier
 4      * valeur absolue flottant
 4      * conversion entier -> flottant
 4      * conversion flottant -> entier
22      * inverse
22      * racine
 4      * ceil
 4      * floor
 2      * goto faux
 2      * goto
 2      * appel
 0      * READ entier
 0      * READ flottant
 0      * READ booleen
 0      * WRITE entier
 0      * WRITE flottant
 0      * WRITE booleen
 0      * pas d'operation
```

INSTRUCTIONS DU PROCESSEUR PARALLELE - SIMD

DUREE	INSTRUCTION
2	* diffusion de scalaire
2	* emission de scalaire vers la file
2	* lecture memoire
2	* ecriture memoire
1	* compression
1	* expansion
1	* bit_reverse
1	* perfect_shuffle
1	* inverse_perfect_shuffle
1	* addition entier
4	* addition flottant
1	* soustraction entier
4	* soustraction flottant
1	* multiplication entier
4	* multiplication flottant
22	* division entier
22	* division flottant
22	* modulo
22	* reste
1	* et
1	* ou
1	* ou exclusif
1	* = entier
4	* = flottant
1	* = booleen
1	* # entier
4	* # flottant
1	* # booleen
1	* < entier
4	* < flottant
1	* > entier
4	* > flottant
1	* <= entier
4	* <= flottant
1	* >= entier
4	* >= flottant
1	* maximum entier
4	* maximum flottant
1	* minimum entier
4	* minimum flottant
1	* merge entier
4	* merge flottant
1	* merge booleen
1	* moins unaire entier
4	* moins unaire flottant
22	* inverse
4	* ceil
4	* floor
1	* non
22	* racine
1	* valeur absolue entier
4	* valeur absolue flottant
1	* sauver dans registre
1	* restaurer le contenu d'un registre

```
1        * echange sommets de pile
4        * conversion entier -> flottant
4        * conversion flottant -> entier
2        * lire adresse
1        * spmd
1        * demasquage
1        * masquage
1        * complementation de masque
0        * READ entier
0        * READ flottant
0        * READ booleen
0        * WRITE entier
0        * WRITE flottant
0        * WRITE booleen
```

INSTRUCTIONS DU PROCESSEUR PARALLELE - SPMD

DUREE	INSTRUCTION
1	* lecture immediate
2	* lecture memoire absolue
2	* lecture memoire basee
2	* lecture memoire indexee
2	* lecture memoire indirecte
2	* lecture memoire indirecte registre
2	* lecture memoire postincrementee
2	* lecture memoire predecrementee
1	* lecture registre
2	* ecriture memoire absolue
2	* ecriture memoire basee
2	* ecriture memoire indexee
2	* ecriture memoire indirecte
2	* ecriture memoire indirecte registre
2	* ecriture memoire postincrementee
2	* ecriture memoire predecrementee
1	* ecriture registre
1	* sauvegarde dans registre
2	* iteration
1	* addition entier
4	* addition flottant
1	* soustraction entier
4	* soustraction flottant
1	* multiplication entier
4	* multiplication flottant
22	* division entier
22	* division flottant
1	* > entier
4	* > flottant
1	* < entier
4	* < flottant
1	* >= entier
4	* >= flottant
1	* <= entier
4	* <= flottant
1	* = entier
4	* = flottant
1	* = booleen
1	* # entier
4	* # flottant
1	* # booleen
1	* minimum entier
4	* minimum flottant
1	* maximum entier
4	* maximum flottant
22	* modulo
22	* reste
1	* et
1	* ou
1	* ou exclusif
1	* echange sommets de pile
1	* moins unaire entier
4	* moins unaire flottant
1	* non

```
1       * valeur absolue entier
4       * valeur absolue flottant
4       * conversion entier -> flottant
4       * conversion flottant -> entier
22      * inverse
22      * racine
4       * ceil
4       * floor
2       * goto faux
2       * goto
2       * appel
0       * READ entier
0       * READ flottant
0       * READ booleen
0       * WRITE entier
0       * WRITE flottant
0       * WRITE booleen
0       * pas d'operation
```

ANNEXE 2

```
programme tred2_vect:

! PRINCIPE DE L'ALGORITHME :
! Tridiagonalisation de la matrice A(n,n) par la methode de Householder
! par la suite de transformations :
!    A:=Pi*A*Pi i=1,n-2;

! La matrice tridiagonale resultat est stockee dans les vecteurs D et E :
!    D=( D(1), D(2), ..., D(n) )
!    E=( 0, E(2), ..., E(n) )


! En sortie,la matrice A est inchangee
!           la matrice Q contient le produit des transformations Pi

! Pour une execution enchainee avec
!   1- tql2_seq.opsila ou tql2_vect1.opsila
!      on doit connaitre (D,E,Q)

! REFERENCES :
!   Bibliotheque EISPACK
!   Wilkinson-Reinsch
!     Handbook for Automatic Computation - Vol 2 - Linear Algebra
!     Springer Verlag - 1971




type
 M: tableau discret*discret de scalaire;
 V: tableau discret de scalaire;

patron
 sous_vecteur (valeur d,f: entier;) de V dans V:
  pour_tout i dans [d,f]:
   (i)
  fin_pour_tout;

 sous_matrice (valeur d1,f1,d2,f2: entier;) de M dans M:
  pour_tout i dans [d1,f1],
  pour_tout j dans [d2,f2]:
   (i,j)
  fin_pour_tout;

 diag de M dans V:
  pour_tout i dans operande.type_.indices.gauche :
   (i,i)
  fin_pour_tout;

variable
 TEMPS,TINIT,CORRECT:entier;
 n:entier;
 zero,un:entier;
 tol,f,g,s,h,l:reel;
 ti:booleen;
 A_100:tableau [1,100]*[1,100] de reel;
 Q_100:tableau [1,100]*[1,100] de reel;
 D_100,E_100,E2_100:tableau [1,100] de reel;
 TEMP,UNITAIRE:tableau [1,100] de reel;
```

```
 TRANSFORM:tableau [1,100] de booleen;

debut

 lire(tol);
 lire(n);

bloc

variable
 A:A_100.sous_matrice(1,n,1,n);
 Q:Q_100.sous_matrice(1,n,1,n);
 D:D_100.sous_vecteur(1,n);
 E:E_100.sous_vecteur(1,n);
 E2:E2_100.sous_vecteur(1,n);

debut

 TEMPS:=date;
 CORRECT:=date-TEMPS;


 lire(A);
 Q:=A;
 TRANSFORM:=vrai;

 zero:=0;
 un:=1;

 ! Determination de la matrice tridiagonale : (D,E)

 TINIT:=date;
 pour i:=n decrement 1 jusqu_a 3
  boucle
   s:=Somme(pour_tout k dans [1,i-2] : Q(i,k)*Q(i,k) fin_pour_tout);
   si s<=tol alors
!   Transformation sautee : Pi=I .
    E(i):=Q(i,i-1);
    pour_tout k dans [1,i-1]:
     Q(i,k):=zero;
     Q(k,i):=zero;
    fin_pour_tout;
    TRANSFORM(i):=faux;
   sinon
!   Tranformation : Q:=Pi*Q*Pi .
    f:=Q(i,i-1);
    s:=s+f*f;
    si f>=zero alors
     g:=-racine_carree(s);
    sinon
     g:=racine_carree(s);
    fin_si;
    E(i):=g;
    h:=s-f*g;
!   On stocke dans la ieme ligne de la partie triangulaire inferieure de Q le ve
cteur Ui
    Q(i,i-1):=f-g;
!   On stocke dans la ieme colonne de la partie
!   triangulaire superieure de Q le vecteur Ui/h
```

```
    pour_tout k dans [1,i-1]:
     Q(k,i):=Q(i,k)/h;
    fin_pour_tout;
!   Calcul de Q*Ui/h dans E
    pour_tout k dans [1,i-1]:
     E(k):=zero;
    fin_pour_tout;
    pour j:=1 jusqu_a i-1
     boucle
      pour_tout k dans [1,i-1]:
       TEMP(k):=si k<=j alors Q(j,k) sinon Q(k,j) fin_si;
       E(k):=E(k)+TEMP(k)*Q(i,j);
      fin_pour_tout;
     fin_boucle;
    pour_tout k dans [1,i-1]:
     E(k):=E(k)/h;
    fin_pour_tout;
    f:=Somme(pour_tout k dans [1,i-1]: E(k)*Q(i,k) fin_pour_tout);
    l:=f/(h+h);
!   Calcul de Q*Ui/h-l*Ui dans E
    pour_tout k dans [1,i-1]:
     E(k):=E(k)-l*Q(i,k);
    fin_pour_tout;
!   Q:=Pi*Q*Pi
    pour j:=1 jusqu_a i-1
     boucle
      pour_tout k dans [1,j]:
       Q(j,k):=Q(j,k)-Q(i,j)*E(k)-Q(i,k)*E(j);
      fin_pour_tout;
     fin_boucle;
   fin_si;
  fin_boucle;

 E(1):=zero;
 E(2):=Q(2,1);
 D:=Q.diag;

! Calcul des vecteurs propres :
! La matrice Q contient le produit des matrices de transformation .

 pour_tout k dans [1,n-1]:
  UNITAIRE(k):=zero;
 fin_pour_tout;
 UNITAIRE(n):=un;

! Q:=P(n-2)
 pour_tout i dans [1,2],
 pour_tout j dans [1,2]:
  Q(i,j):=-Q(i,3)*Q(3,j);
 fin_pour_tout;
 Q.diag.sous_vecteur(1,2):=un+Q.diag.sous_vecteur(1,2);
 pour_tout k dans [1,3]:
  Q(k,3):=zero;
  Q(3,k):=UNITAIRE(n-3+k);
 fin_pour_tout;

! Q:=Q*Pi  i=n-3,...,1
 pour i:=4 jusqu_a n
  boucle
   ti:=TRANSFORM(i);
```

```
    si ti alors
     pour_tout k dans [1,i-2]:
      TEMP(k):=zero;
     fin_pour_tout;
     pour j:=1 jusqu_a i-1
      boucle
       pour_tout k dans [1,i-2]:
        TEMP(k):=TEMP(k)+Q(i,j)*Q(j,k);
       fin_pour_tout;
      fin_boucle;
     TEMP(i-1):=Q(i,i-1);
     pour_tout j dans [1,i-1],
     pour_tout k dans [1,i-1]:
      Q(k,j):=Q(k,j)-TEMP(j)*Q(k,i);
     fin_pour_tout;
    fin_si;
    pour_tout k dans [1,i]:
     Q(k,i):=zero;
     Q(i,k):=UNITAIRE(n-i+k);
    fin_pour_tout;
   fin_boucle;


  TEMPS:=date-TINIT-CORRECT;

  ecrire(TEMPS);


  ecrire(D);
  ecrire(E);
  ecrire(Q);

 fin;

 fin.
```

ANNEXE 3

```
programme tql2_vect :

! Diagonalisation d'une matrice tridiagonale d'ordre n : (D,E)
! par l'algorithme QL.

! Les valeurs et vecteurs propres de la matrice sont calculees.


! Si la matrice tridiagonale est obtenue par l'algorithme tred2_vect.opsila
! la matrice VECT contient, en entree, le produit des transformations de
! Householder, en sortie les vecteurs propres de la matrice d'entree dans tred2_
vect.

! On travaille sur les vecteurs D et E


!   PRINCIPE DE L'ALGORITHME :
! On annul e successivement les elements hors-diagonaux.
! Soit  binf le rang du premier element hors-diagonal non nul ,
!       bsup le rang du premier element hors-diagonal nul apres binf.
! On applique les iterations QL sur le bloc delimite par les lignes (binf,bsup),
!pour annul er l'element de rang binf.
! Les iterations sont vectorielles.


type
 V:tableau discret de scalaire;
 M:tableau discret*discret de scalaire;

patron
 sous_vecteur (valeur d,f: entier;) de V dans V:
  pour_tout i dans [d,f]:
   (i)
  fin_pour_tout;
 sous_matrice (valeur d1,f1,d2,f2: entier;) de M dans M:
  pour_tout i dans [d1,f1],
  pour_tout j dans [d2,f2]:
   (i,j)
  fin_pour_tout;

variable
 TEMPS,TINIT,CORRECT:entier;
 zero,un,deux,quatre:entier;
 n:entier;
 binf,bsup:entier;
 nb_iter:entier;
 macheps:reel;
 f,g,h,shift,shift1,shift2,delta:reel;
 d_inf,d_inf1,e_inf:reel;
 puissance:entier;
 t_ql,quot,log1,log2:entier;
 fail:booleen;
 TEMPO:tableau [1,1] de reel;
 D_100,E_100,SAUVE:tableau [1,100] de reel;
 VECT_100:tableau [1,100]*[1,100] de reel;
 R2,SIN,COS:tableau [1,100] de reel;
 W,Z2,A:tableau [1,101] de reel;
 B,C:tableau [1,100]*[1,4] de reel;
```

```
debut

 TEMPS:=date;
 CORRECT:=date-TEMPS;

 lire(macheps);
 lire(n);

 bloc

 variable
  D:D_100.sous_vecteur(1,n);
  E:E_100.sous_vecteur(1,n);
  VECT:VECT_100.sous_matrice(1,n,1,n);

 debut

! D : vecteur des elements diagonaux;
! E : vecteur des elements hors-diagonaux;
! VECT : matrice des vecteurs propres;
  lire(D);
  lire(E);
  lire(VECT);

  zero:=0;
  un:=1;
  deux:=2;
  quatre:=4;

  pour_tout k dans [2,n]:
   E(k-1):=E(k);
  fin_pour_tout;
  E(n):=zero;
  f:=zero;
  g:=zero;
  fail:=faux;

  bloc

  etiquette
   B1;

  debut

   TINIT:=date;

   binf:=1;
   ?B1
   boucle
!   QL sur le bloc debutant a la ligne binf
!                  finissant a la ligne bsup  ; bsup>binf;
!    binf est le rang du premier element hors-diagonal non nul;
!    bsup est le rang du premier element hors-diagonal nul rencontre a partir de
 la ligne binf;
    si binf>n alors
     sortir_de B1;
    fin_si;
    si fail alors
     sortir_de B1;
    fin_si;
```

```
    nb_iter:=0;
    d_inf:=D(binf);
    e_inf:=E(binf);
    h:=macheps*(absolu(d_inf)+absolu(e_inf));
    si g<h alors
     g:=h;
    fin_si;
    bsup:=Min(pour_tout j dans [binf,n]:
              si absolu(E(j))<=g alors
               j
              sinon
               n
              fin_si
             fin_pour_tout);
    si bsup>binf alors
     t_ql:=bsup-binf+1;

     bloc

     etiquette
      LOG1;

     debut

!     Calcul de log1=[log2(t_ql)]
      log1:=0;
      quot:=t_ql-1;
      ?LOG1
      boucle
       si quot=0 alors
        sortir_de LOG1;
       fin_si;
       log1:=log1+1;
       quot:=quot div 2;
      fin_boucle;

     fin;

     si t_ql>2 alors

      bloc

      etiquette
       LOG2;

      debut

!      Calcul de log2=[log2(t_ql-2)]
       log2:=0;
       quot:=t_ql-3;
       ?LOG2
       boucle
        si quot=0 alors
         sortir_de LOG2;
        fin_si;
        log2:=log2+1;
        quot:=quot div 2;
       fin_boucle;

      fin;
```

```
fin_si;

bloc

etiquette
 ITERATION;

debut

 ?ITERATION
 boucle
  si nb_iter=30 alors
   fail:=vrai;
   sortir_de ITERATION;
  fin_si;
  nb_iter:=nb_iter+1;
!     Calcul du shift
  d_inf1:=D(binf+1);
  delta:=(d_inf1-d_inf)*(d_inf1-d_inf)+quatre*e_inf*e_inf;
  shift1:=(d_inf+d_inf1-racine_carree(delta))/deux;
  shift2:=(d_inf+d_inf1+racine_carree(delta))/deux;
  si absolu(d_inf-shift1)<absolu(d_inf-shift2) alors
   shift:=shift1;
  sinon
   shift:=shift2;
  fin_si;
  pour_tout i dans [binf,n]:
   D(i):=D(i)-shift;
  fin_pour_tout;
  f:=f+shift;

  si t_ql>2 alors
!      Resolution du systeme:
!       W(j)=D(j)*W(j+1)-E(j)*E(j)*W(j+2)  j=binf+1,bsup-1
!       W(bsup)=D(bsup)
!       W(bsup+1)=1
   pour_tout j dans [binf+1,bsup-1]:
    B(j,1):=D(j);
    B(j,2):=-E(j)*E(j);
    B(j,3):=un;
    B(j,4):=zero;
   fin_pour_tout;
   puissance:=1;
   pour i:=1 jusqu_a log2
   boucle
    pour_tout j dans [binf+1,bsup-1-puissance]:
     C(j,1):=B(j,1)*B(j+puissance,1)+B(j,2)*B(j+puissance,3);
     C(j,2):=B(j,1)*B(j+puissance,2)+B(j,2)*B(j+puissance,4);
     C(j,3):=B(j,3)*B(j+puissance,1)+B(j,4)*B(j+puissance,3);
     C(j,4):=B(j,3)*B(j+puissance,2)+B(j,4)*B(j+puissance,4);
    fin_pour_tout;
    pour_tout k dans [1,4],
    pour_tout j dans [binf+1,bsup-1-puissance]:
     B(j,k):=C(j,k);
    fin_pour_tout;
    puissance:=puissance*2;
   fin_boucle;
   pour_tout j dans [binf+1,bsup-1]:
    W(j):=B(j,1)*D(bsup)+B(j.2);
```

```
           fin_pour_tout;
          fin_si;
          W(bsup):=D(bsup);
          W(bsup+1):=un;
!         Resolution du systeme:
!          Z2(j)=E(j-1)*E(j-1)*Z2(j+1)+W(j)*W(j)   j=binf+1,bsup
!          Z2(bsup+1)=1
          pour_tout j dans [binf+1,bsup+1]:
           Z2(j):=W(j)*W(j);
          fin_pour_tout;
          pour_tout j dans [binf+1,bsup]:
           A(j):=E(j-1)*E(j-1);
          fin_pour_tout;
          A(bsup+1):=un;
          puissance:=1;
          pour i:=1 jusqu_a log1
          boucle
           pour_tout j dans [binf+1,bsup+1-puissance]:
            Z2(j):=Z2(j)+A(j)*Z2(j+puissance);
            A(j):=A(j)*A(j+puissance);
           fin_pour_tout;
           puissance:=puissance*2;
          fin_boucle;
!         Calcul des elements diagonaux et hors-diagonaux du bloc [binf...bsup]
          pour_tout j dans [binf+1,bsup]:
           R2(j):=Z2(j)/Z2(j+1);
           SIN(j-1):=E(j-1)/racine_carree(R2(j));
           COS(j-1):=W(j)/racine_carree(Z2(j));
          fin_pour_tout;
          SIN(bsup):=zero;
          pour_tout j dans [binf,bsup]:
           SAUVE(j):=D(j);
          fin_pour_tout;
          pour_tout j dans [binf+1,bsup]:
           D(j):=(W(j)*W(j+1)*(R2(j)+E(j-1)*E(j-1))/Z2(j))+SIN(j-1)*SIN(j-1)*SAUVE(
j-1);
          fin_pour_tout;
          pour_tout j dans [binf+1,bsup]:
           E(j):=SIN(j)*racine_carree(R2(j));
          fin_pour_tout;
          TEMPO(1):=W(binf+1)*D(binf)-W(binf+2)*E(binf)*E(binf);
          D(binf):=W(binf+1)*TEMPO(1)/Z2(binf+1);
          E(binf):=SIN(binf)*TEMPO(1)/racine_carree(Z2(binf+1));

!         Calcul de la matrice des vecteurs propres
          pour i:=bsup-1 decrement 1 jusqu_a binf
          boucle
           pour_tout j dans [1,n]:
            SAUVE(j):=VECT(j,i+1);
            VECT(j,i+1):=SIN(i)*VECT(j,i)+COS(i)*SAUVE(j);
            VECT(j,i):=COS(i)*VECT(j,i)-SIN(i)*SAUVE(j);
           fin_pour_tout;
          fin_boucle;

          e_inf:=E(binf);
          d_inf:=D(binf);
          si absolu(e_inf)<=g alors
           sortir_de ITERATION;
          fin_si;
         fin_boucle; !ITERATION
```

```
      fin;

     fin_si;
     D(binf):=D(binf)+f;
     binf:=binf+1;
    fin_boucle; !B1

    TEMPS:=date-TINIT-CORRECT;

   fin;

   si fail alors
    D:=zero;
   fin_si;
   ecrire(TEMPS);
   ecrire(D);
   ecrire(VECT);

  fin;

 fin.
```

ANNEXE 4

```
programme TREPS_SPMD :

constante
  nproc : 8 ; ordre : 64 ; ordrew : ordre ;
  ordrewsp : ordrew div nproc + 1 ; nbloc : ordre ;
variable
  a, b, eps, diag, sdiag : reel ;
  n, m, nsub, ierr : entier ;
  D, E, E2, RV1, RV2, RV3 : tableau [1, ordre] de reel ;
  W : tableau [1, ordrew] de reel ;
  IV : tableau [1, ordre] de entier ;
  STK : tableau [1, ordrew] de entier ;
  LROW, RANKF, NEVSUB : tableau [1, nbloc] de entier ;



! ====================== debut TREPS ======================================

procedure TREPS (valeur
                   n : entier ;
                   bg, bd : reel ;
                   D : tableau un_entier de reel ;
                   E : tableau D.type_.indices de reel ;
                 variable
                   m, nsub, ierr : entier ;
                   eps : reel ;
                   E2 : tableau D.type_.indices de reel ;
                   IV : tableau D.type_.indices de entier ;
                   W : tableau un_entier de reel ;
                   STK: tableau W.type_.indices de entier ;
                   LROW : tableau un_entier de entier ;
                   RANKF, NEVSUB : tableau LROW.type_.indices de entier ;
                   RV1, RV2, RV3 : tableau D.type_.indices de reel ;
                ) :

etiquette
  par_bloc ; retour ;
variable
  SECT : tableau [1, nproc+2] de entier ;
  RANG : tableau [1, nproc] de entier ;
spmd
constante
  nproc : 8 ; ordre : 64 ; ordrew : ordre ;
  ordrewsp : ordrew div nproc + 1 ; nbloc : ordre ;
  variable
    D, E, E2 : tableau [1,ordre] de reel ;
    BG, BD : tableau [1,ordrewsp] de reel ;
    FSTEV : tableau [1,ordrewsp] de entier ;
    p, q, top1, bidon : entier ;
    eps1 : reel ;
    zero, un, deux, trois, point5, epsilon, petit : reel ;
fin_spmd ;
 variable
  p, q, ancm, isl, isu, nvp, top1, top2, grd2, nvpi, mm, mxsub,
     nprocp1, nprocm1, qt, r,
     mult, itrie, nssint, iliste, ivar, ider, rangi : entier ;
  lb, ub, var1, var2, h, lb0, ub0, epsilon, zero, lhi,
                   un, deux, trois, point5, eps1, petit : reel ;
  premier : booleen ;
 patron
```

```
  INTERNE de tableau [1,nproc+2] de entier
          dans tableau [1,nproc] de entier :
     pour_tout i dans [1, nproc] : (i + 1) fin_pour_tout ;

variable
  BG : pour_tout i dans [1,ordrewsp*nproc] :
          en_spmd`BG (i-1) (1)
       fin_pour_tout ;
  BD : pour_tout i dans [1,ordrewsp*nproc] :
          en_spmd`BD (i-1) (1)
       fin_pour_tout ;
  FSTEV : pour_tout i dans [1,ordrewsp*nproc] :
          en_spmd`FSTEV (i-1) (1)
       fin_pour_tout ;
procedure

!======================= ERREUR =====================================

 ERREUR (valeur nerr : entier ; variable ierr : entier ;) :
  debut
   ierr := nerr ;
   ecrire (- nerr) ;
  fin ;

!======================= DIFFUSER ==================================
 DIFFUSER_R (valeur n : entier ;
                    T : tableau [1,n] de reel ;
            variable  EXT : tableau [0, nproc-1] de
                           tableau [1,n] de reel ;) :
   debut
     pour_tout j dans [1,n],
     pour_tout k dans [0,nproc-1] :
       EXT(k)(j) := T(j) ;
     fin_pour_tout ;
   fin ;

!======================= MUSECT =====================================

 MUSECT (valeur lb, ub : reel ; neval : entier ;
                  variable S : tableau un_entier de entier ;)  :
     ! Cette procedure calcule la suite de Sturm
     ! en neval points regulierement espaces a
     ! l'interieur de l'intervalle [lb,ub].
  variable
    U, V : tableau [1,nproc] de reel ;
    h : reel ;
  debut
    h := (ub - lb) / (neval + 1) ;
    pour_tout k dans [1,neval] :
      V(k) := lb + k * h ;
      U(k) := D(p) - V(k) ;
      S(k) := si (U(k) < zero) alors 1 sinon 0 fin_si ;
    fin_pour_tout;
    pour i:= p+1 increment jusqu_a q
      boucle
        si (Union(U=zero)) alors
         pour_tout k dans [1, neval] :
          U(k) := D(i) - V(k) -  (si U(k)=zero alors absolu(E(i))/epsilon
                                           sinon E2(i)/U(k)
                                 fin_si);
```

```
          si (U(k) < zero) alors S(k) := S(k) + 1 ; fin_si ;

        fin_pour_tout ;
                          sinon
        pour_tout k dans [1, neval] :
          U(k) := D(i) - V(k) - E2(i)/U(k) ;
          si (U(k) < zero) alors S(k) := S(k) + 1 ; fin_si ;
        fin_pour_tout ;
       fin_si ;
     fin_boucle ;
 fin ;


!====================== STURM ==============================

 STURM(valeur v : reel ; variable s : entier ; ) :
 variable u : reel ;
 debut
  u := D(p) - v ;
  s := si u < zero alors 1 sinon 0 fin_si ;
  pour i := p+1 increment jusqu_a q
   boucle
     si u = zero alors u := absolu(E(i))/epsilon ;
                 sinon u := E2(i)/u ;
     fin_si ;
    u := D(i) - v - u ;
    si u < zero alors s := s+1 ; fin_si ;
   fin_boucle ;
  fin ;



!====================== EXTRAIRE =======================

spmd procedure
  EXTRAIRE (variable rang : entier ; bg, bd : reel ;) :
    variable x, a, b, c, fa, fb, fc, d, e, tol1,
                         xm, pr, qr, rr, sr : reel ;
             s : entier ;
    etiquette bisections ; retour_extraire ;
    procedure
      STURM_VAL(valeur v : reel ; variable w : reel ; ) :
        variable u : reel ;
        debut
          u := D(p) - v ; w := u ;
          pour i := p+1 increment jusqu_a q
           boucle
             si u = zero alors u := absolu(E(i))/epsilon ;
                         sinon u := E2(i)/u ;
             fin_si ;
            u := D(i) - v - u ; w := w * u ;
           fin_boucle ;
          fin ;
      STURM(valeur v : reel ; variable s : entier ; ) :
        variable u : reel ;
        debut
          u := D(p) - v ;
          s := si u < zero alors 1 sinon 0 fin_si ;
          pour i := p+1 increment jusqu_a q
           boucle
```

```
        si u = zero alors u := absolu(E(i))/epsilon ;
                    sinon u := E2(i)/u ;
        fin_si ;
       u := D(i) - v - u ;
       si u < zero alors s := s+1 ; fin_si ;
      fin_boucle ;
     fin ;
debut
  a := bg ; b := bd ;
  STURM_VAL(a,fa) ; STURM_VAL(b,fb) ; fc := fb ;
  boucle
  bloc
   etiquette une_bisection ; fin_etape ;
  debut
     si (fb*fc > zero ) alors
       c := a ; fc := fa ; d := b-a ; e := d ;
     fin_si ;
     si (absolu(fc) < absolu(fb)) alors
       a := b ; b := c ; c:= a ;
       fa := fb ; fb := fc ; fc := fa ;
     fin_si ;
  ! test de convergence
     tol1 := deux * epsilon * absolu(b) + point5 * eps1 ;
     xm := point5 * (c - b) ;
     si ( absolu(xm) < tol1 ) alors
                 bg := b; aller_a retour_extraire ; fin_si ;
     si (absolu(fb) < petit ) alors aller_a bisections ; fin_si ;
  ! a cette etape une bisection est-elle necessaire ?
     si (absolu(e) < tol1) ou (absolu(fa) < absolu(fb)) alors
       aller_a une_bisection ;
     fin_si ;
  ! interpolation quadratique ou lineaire si impossible
     si ( a = c ) alors
       sr := fb/fa ; pr := deux * xm * sr ;
       qr := un - sr ;
                 sinon
       qr := fa/fc ; rr := fb/fc ; sr := fb/fa ;
       pr := sr * (deux*xm*qr*(qr-rr) - (b-a)*(rr-un)) ;
       qr := (qr-un) * (rr-un) * (sr-un) ;
     fin_si ;
  ! controle des signes
     si pr > zero alors qr := - qr ; fin_si ;
     pr := absolu(pr) ;
  ! l'interpolation est-elle acceptable ?
    si ( ( (deux*pr) >= (trois*xm*qr - absolu(tol1*qr)) )
                          ou
        ( pr >= absolu(point5*e*qr) ) ) alors
      aller_a une_bisection ;
    fin_si ;
    e := d ; d := pr/qr ; aller_a fin_etape ;
  !
   ?une_bisection d := xm ; e := d ;
   ?fin_etape
    a := b ; fa := fb ;
    si (absolu(d) > tol1) alors b := b + d ;
                        sinon b := (si xm > zero
                                        alors tol1
                                        sinon -tol1
                                        fin_si) + b ;
    fin_si ;
```

```
          STURM_VAL(b,fb) ;
        fin ;
        fin_boucle ;
       ?bisections
           bg := min(a,min(b,c)) ; bd := max(a,max(b,c)) ;
         boucle
           x := point5 * (bg + bd) ;
           si ( bd - bg < deux * epsilon * (absolu(bg) + absolu(bd))
                                             + point5 * eps1  )
             alors aller_a retour_extraire ; fin_si ;
           STURM(x, s) ;
           si rang > s alors bg := x ;
                        sinon bd := x ;
           fin_si ;
         fin_boucle ;
       ?retour_extraire bidon := 0 ;
      fin ;
    fin_spmd ;

!====================== DEBUT DU CORPS DE TREPS ==============================

debut

! definition des tailles de tableaux

nprocm1 := nproc - 1 ; nprocp1 := nproc + 1 ;
mm := W.type_.indices.cardinal ;
mxsub := LROW.type_.indices.cardinal ;
si (n > D.type_.indices.cardinal) alors
              ERREUR(3,ierr) ; aller_a retour ; fin_si ;

! initialisations

lire(zero, un, deux, trois, point5, epsilon, petit) ;
pour_tout k dans [0,nproc-1] :
  en_spmd`zero(k) := zero ;
  en_spmd`un(k) := un ;
  en_spmd`deux(k) := deux ;
  en_spmd`trois(k) := trois ;
  en_spmd`point5(k) := point5 ;
  en_spmd`epsilon(k) := epsilon ;
  en_spmd`petit(k) := petit ;
fin_pour_tout ;
DIFFUSER_R(n, D, en_spmd`D) ;
DIFFUSER_R(n, E, en_spmd`E) ;
DIFFUSER_R(n, E2, en_spmd`E2) ;
ierr := 0 ; m := 0 ; nsub := 0 ;
top1 := 0 ; grd2 := mm + 1 ;
pour_tout i dans [1,n] :
  RV1(i) := absolu(E(i)) ;
  RV2(i) := absolu(D(i)) ;
fin_pour_tout ;
pour_tout i dans [2,n] :
  RV3(i) := RV2(i) + RV2(i-1) ;
fin_pour_tout ;

! isolement des valeurs propres

q := 0 ; ancm := 0 ;
?par_bloc
```

```
boucle
 bloc
                                                    ! boucle par_bloc
 .
 etiquette
  limite_bloc ; fusion ; fin_fusion ;
  isolement  ;
 debut
  p := q+1;
  si p>n alors sortir_de par_bloc ; fin_si ;
  nsub := nsub+1 ;
  si (nsub > mxsub) alors ERREUR(2,ierr) ; aller_a retour ; fin_si ;
  ?limite_bloc
  boucle
    q := q+1 ;
    si q=n alors sortir_de limite_bloc ; fin_si ;
    si ( (RV3(q+1) + RV1(q+1)) * epsilon > RV1(q+1) )
      alors  E2(q+1) := zero ; sortir_de limite_bloc ;
    fin_si ;
  fin_boucle ;
  pour_tout k dans [0,nproc-1] :
   en_spmd`p(k) := p ; en_spmd`q(k) := q ;
  fin_pour_tout ;

  ! traitement du cas d'element isole

  si q=p alors
    si ( (D(p) >= bg) et (D(p) <= bd) )
      alors
        m := m + 1 ;
        si m>mm alors ERREUR(1,ierr) ; aller_a retour ; fin_si ;
        W(p) := D(p) ; IV(p) := nsub ;
        NEVSUB(nsub) := 1 ; LROW(nsub) := p ; RANKF(nsub) := p ;
      sinon
        reboucler par_bloc ;
    fin_si ;
    aller_a fusion ;
  fin_si ;

  ! calcul du domaine de Gerschgorin

  pour_tout i dans [p,q-1]:
    RV2(i) := RV1(i) + RV1(i+1) ;
  fin_pour_tout ;
  RV2(q) := RV1(q) ;
  pour_tout i dans [p,q] :
    RV3(i) := D(i) - RV2(i) ;
  fin_pour_tout ;
  lb := Min(pour_tout i dans [p,q] : RV3(i) fin_pour_tout ) ;
  pour_tout i dans [p,q] :
    RV3(i) := RV2(i) + D(i) ;
  fin_pour_tout ;
  ub := Max(pour_tout i dans [p,q] : RV3(i) fin_pour_tout ) ;
  eps1 := max(absolu(lb),absolu(ub))*(si eps < zero alors epsilon
                                                sinon eps fin_si ) ;
  pour_tout k dans [0,nproc-1] :
    en_spmd`eps1(k) := eps1 ;
  fin_pour_tout ;
  lb := max(lb,bg) ;
  ub := min(ub,bd) ;
```

```
! premiere multisection

  h := (ub - lb) / nprocm1 ; ubO := ub + h ; lbO := lb - h ;
  MUSECT(lbO, ubO, nproc, SECT) ;
  nvp := SECT(nproc) - SECT(1) ; NEVSUB(nsub) := nvp ;
  si (nvp = 0 ) alors reboucler par_bloc ; fin_si ;
  m := m + nvp ; si (m > mm) alors ERREUR(1,ierr) ; aller_a retour ; fin_si ;
  RANKF(nsub) := SECT(1) + p ;
  pour i := 1 jusqu_a nproc - 1
    boucle
     nvpi := SECT(i+1) - SECT(i) ;
     si (nvpi <> 0) alors
      lhi := lb + i * h ;
      si (nvpi = 1) alors top1 := top1 + 1 ; STK(top1) := nsub ;
                     FSTEV(top1) := SECT(i) + 1 ;
                     BG(top1) := lhi - h ; BD(top1) := lhi ;
                   sinon grd2 := grd2 - 1 ; STK(grd2) := nvpi ;
                     FSTEV(grd2) := SECT(i) + 1 ;
                     BG(grd2) := lhi - h ; BD(grd2) := lhi ;
      fin_si ;
     fin_si ;
    fin_boucle ;
! isolement de toutes les valeurs propres du bloc

?isolement
boucle      !tant qu'il reste des valeurs propres a isoler
bloc
etiquette partition ;
debut

 si (grd2 = mm + 1) alors sortir_de isolement ; fin_si ;
 SECT(1) := FSTEV(grd2) - 1 ; SECT(nproc+2) := SECT(1) + STK(grd2) ;
 lb := BG(grd2) ; ub := BD(grd2) ;
 grd2 := grd2 +1 ;

   ?partition
   boucle       !jusqu'a ce qu'aucun des sous-intervalles
                !  ne contienne plus d'une valeur propre

    ! calcul de l'ordre de la multisection
    var1 := nprocp1 * (absolu(lb) + absolu(ub)) ;
    var2 := var1 + (ub - lb) ;
    si (var1 = var2 )
      alors mult := en_entier( (ub - lb) / eps1 ) ;
            si (mult <= 0 )
              alors isl := SECT(1) ; isu := SECT(nproc+2) - 1 ;
                    pour_tout i dans [isl, isu] :
                      W(p+i) := lb ; IV(p+i) := nsub ;
                    fin_pour_tout ;
                    sortir_de partition ;
              sinon mult := min(nproc, mult) ;
                    SECT(mult+2) := SECT(nproc+2) ;
            fin_si ;
      sinon mult := nproc ;
    fin_si ;

    ! multisection
    MUSECT(lb, ub, mult, SECT.INTERNE) ;
    premier := vrai ; h := (ub - lb) / (mult + 1) ;
    pour i := 1 jusqu_a mult + 1
```

```
     boucle
      nvpi := SECT(i+1) - SECT(i) ;
      si nvpi <> 0 alors
       lhi := lb + i * h ;
       si (nvpi = 1) alors top1 := top1 + 1 ; STK(top1) := nsub ;
                      FSTEV(top1) := SECT(i) + 1 ;
                      BG(top1) := lhi - h ; BD(top1) := lhi ;
                    sinon
                      si (premier)
                       alors premier := faux ; nssint := i ;
                       sinon grd2 := grd2 - 1 ; STK(grd2) := nvpi ;
                             FSTEV(grd2) := SECT(i) + 1 ;
                             BG(grd2) := lhi - h ; BD(grd2) := lhi ;
                      fin_si ;
       fin_si ;
      fin_si ;
     fin_boucle ;
    si (premier) alors sortir_de partition ; fin_si ;

    ub := lb + nssint * h ; lb := ub - h ;
    ivar := SECT(nssint) ; SECT(nproc+2) := SECT(nssint+1) ;
    SECT(1) := ivar ;

   fin_boucle ;  ! boucle partition
fin ;


fin_boucle ;  ! boucle isolement

!extraction des valeurs propres

   ! calcul du nombre de valeurs propres par processeur
     qt := top1 div nproc ; r := top1 - qt * nproc ;
     pour_tout k dans [0,nproc-1] :
       en_spmd`top1(k) := si k < r alors qt + 1
                                    sinon qt
                          fin_si ;
     fin_pour_tout ;
   ! extraçtion parallele
 spmd
     pour j := 1 jusqu_a top1
       boucle
         EXTRAIRE(FSTEV(j), BG(j), BD(j) ) ;
       fin_boucle ;
 fin_spmd ;
   ! rangement des valeurs propres
     iliste := RANKF(nsub) ;
     pour j := 1 jusqu_a top1
       boucle
         W(iliste+FSTEV(j)-1) := BG(j) ;
       fin_boucle ;

!fusion de la liste avec les precedentes
?fusion
si (ancm = 0) alors si (RANKF(1)=1) alors
                           pour_tout i dans [1, m] :
                            IV(i) := 1 ;
                           fin_pour_tout ;
                           ancm := m ;
                                      sinon
```

```
                              iliste := RANKF(nsub) - 1 ;
                              pour_tout i dans [1, m] :
                               W(i) := W(iliste+i) ; IV(i) := nsub ;
                              fin_pour_tout ;
                              ancm := m ;
                         fin_si;
                  sinon
    bloc
    etiquette sortie ;
    debut
                         itrie := 1 ; iliste := RANKF(nsub) ;
                         ider := iliste + NEVSUB(nsub) - 1 ;
    boucle
      si (iliste > ider) alors aller_a fin_fusion ; fin_si ;
      si (itrie > ancm ) alors aller_a sortie ; fin_si ;
      si (W(itrie) > W(iliste)) alors
         var1 := W(iliste) ; ancm := ancm + 1 ;
         pour_tout i dans [itrie, ancm -1] :
           W(itrie + ancm - i) := W(itrie + ancm - i-1) ;
           IV(itrie + ancm - i) := IV(itrie + ancm - i-1) ;
         fin_pour_tout ;
         W(itrie) := var1 ; IV(itrie) := nsub ;
         iliste := iliste + 1 ;
      fin_si ;
      itrie := itrie + 1 ;
    fin_boucle ;
   ?sortie
    pour j := itrie increment jusqu_a ider
      boucle
        ancm := ancm + 1 ;
        W(ancm) := W(j) ; IV(ancm) := nsub ;
      fin_boucle ;
  fin ;
  fin_si ;

?fin_fusion top1 := 0 ;
fin ;
fin_boucle ;                                         !   boucle par_bloc

?retour top1 := 0 ;

fin ;
! ====================== fin TREPS ========================================


debut
  lire (D, E) ;
  E2 := E*E ;
  lire (n, a, b, eps) ;
  si n <= ordre alors

   TREPS ( n, a, b, D, E, m, nsub, ierr,
           eps, E2, IV, W, STK,
           LROW, RANKF, NEVSUB, RV1, RV2, RV3 ) ;

   ecrire(m, nsub, ierr) ;
   ecrire(W) ;
   ecrire(IV) ;

                sinon
```

```
    ierr := 10 ; ecrire(ierr) ;
  fin_si ;
fin.
```

ANNEXE 5

Matrices tridiagonales symétriques de test

Deux cas extrêmes :

matrice à valeurs propres bien réparties

matrice à groupe de valeurs propres numériquement confondues.

1 - Matrice L_n = [-1, 2, -1]

ordre n

diagonale = (2, ..., 2)

sous diagonale = (-1, ..., -1)

valeurs propres : $\lambda_k = 2(1 - \cos \frac{k\pi}{n+1})$ k = 1, ..., n

Le spectre est inclus dans l'intervalle [0,4].

2 - Matrice de Wilkinson : W_{21}^{+} [Wi 65 p 308]

ordre 21

diagonale (10, 9, ..., 3, 2, 1, 0, 1, 2, ..., 10)

sous diagonale (1, 1, ..., 1)

valeurs propres :

Eigenvalues of W_{21}^{+}

10·74619 42	5·00024 44
10·74619 42	4·99978 25
9·21067 86	4·00435 40
9·21067 86	3·99604 82
8·03894 11	3·04309 93
8·03894 11	2·96105 89
7·00395 22	2·13020 92
7·00395 18	1·78932 14
6·00023 40	0·94753 44
6·00021 75	0·25380 58
	−1·12544 15

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique