PHYSIQUE THEORIQUE
SACLAY
CEA/DSM

CEA-DSM. T98/075

Calcul de chaînes de spins quantiques sur ordinateur parallèle

J. Lamarcq
CEA/Saclay, Service de Physique Théorique
F-91191 Gif-sur-Yvette Cedex, FRANCE

Publication : Rapport de stage de fin d'études à Supélec, avril-juin 1998

Sous la direction de M. O. Golinelli

Please be aware that all of the Missing Pages in this document were originally blank pages

Julien Lamarcq
IIC

Avril-Juin 1998

CALCUL DE CHAÎNES DE SPINS QUANTIQUES SUR ORDINATEUR PARALLÈLE

Rapport de stage de fin d'études à Supélec effectué au Service de Physique Théorique du CEA sous la direction de M. Olivier Golinelli

Table des matières

Avant-Propos 3

Introduction 5

1 Les systèmes de spins 7
- 1.1 Les origines du magnétisme . . . 7
- 1.2 Un peu de formalisme quantique . . . 8
- 1.3 Les modèles de chaînes de spins . . . 9

2 Le calcul de valeurs propres 11
- 2.1 La méthode de Lanczos . . . 11
- 2.2 Utilisation de la méthode . . . 12
- 2.3 Un peu d'analyse . . . 12

3 Le programme 15
- 3.1 Présentation générale . . . 15
- 3.2 Version de test . . . 16
- 3.3 Version d'exploitation . . . 18

4 La parallélisation 23
- 4.1 Environnement . . . 23
- 4.2 Première parallélisation . . . 24
- 4.3 Seconde parallélisation . . . 25

Conclusion 27

Bibliographie 29

A La librairie NAG 31

B Le protocole SHMEM 33

C Programme parallèle 1 35

D Programme parallèle 2 37

Avant-Propos

Il s'agit de présenter le développement et les résultats d'un programme de simulation numérique, réalisé au Service de Physique Théorique au CEA dans le cadre du stage de fin d'études à Supélec. Ce stage doit être l'aboutissement de deux années d'études en école d'ingénieurs, dont une a été consacrée à l'informatique. Au-delà de la mise en pratique des techniques apprises en contexte scolaire, c'est l'occasion pour moi de travailler en physique théorique, domaine dans lequel je désire poursuivre mes études. Le service est constitué d'environ soixante chercheurs, d'une dizaine de thésards et d'une trentaine de visiteurs. L'informatique est à base de stations Sun sous UNIX et de terminaux X ; la plupart des calculs sont assurés par des serveurs Sun SparcStation. Seuls les gros calculs sont effectués sur un Cray C94 (vectoriel) et un Cray T3E (parallèle) situés à Grenoble.

Ce stage a été effectué sous la direction de Olivier GOLINELLI, qui, outre la chaleur de son accueil et sa disponibilté, a toujours su m'expliquer clairement une physique jamais triviale et me guider dans l'apprentissage du FORTRAN et l'utilisation des ordinateurs Cray. Je l'en remercie vivement.

Introduction

L'informatique a pris — en trois temps — une place très importante en physique. C'est d'abord le calcul scientifique qui ouvrit la marche, suivi une vingtaine d'années plus tard par l'échange des données à travers les réseaux, puis, encore récemment, par l'édition scientifique (formateurs de texte scientifique et serveurs de preprints). La simulation numérique permet souvent au théoricien de se convaincre des modèles qu'il utilise. En particulier, en simulant les chaînes de spins, on peut décider de la justesse ou non d'une conjecture. Mais si la complication des résultats de la physique moderne rend cet outil précieux, sa manipulation devient de moins en moins aisée. En effet, simuler un système à grand nombre de degrés de liberté exige le recours à des machines de plus en plus sophistiquées. Nous allons nous intéresser au domaine de la physique de la matière condensée, et plus particulièrement aux chaînes de spins quantiques. Elles permettent — entre autres — la modélisation des interactions magnétiques entre les ions d'un cristal. Si beaucoup de résultats exacts ont été trouvés pour des systèmes de spin $\frac{1}{2}$, il n'en est pas de même pour d'autres valeurs de spins, pour lesquelles on a effectué de nombreuses simulations. L'intérêt pour les simulations a été surtout relancé après la conjecture de Haldane, stipulant l'existence d'un saut d'énergie entre le niveau fandamental et les premiers niveaux excités d'une chaine de spins 1. L'existence de ce *gap* a depuis été démontrée expérimentalement.

Dans le formalisme quantique, trouver les énergies possibles d'un système donné requiert l'extraction des valeurs propres de l'hamiltonien (i.e. l'opérateur représentant l'énergie de ce système), les états stationnaires correspondants étant les vecteurs propres associés. Ainsi, nous sommes confrontés à un calcul de diagonalisation. Mais il y a deux difficultés particulières : d'une part, la matrice hamiltonienne est énorme, et d'autre part, elle est très creuse, sans qu'il apparaisse de structure simple. En effet, dans le cas de spins $\frac{1}{2}$, où chaque ion n'a que deux états de spin, il y a 2^n états stationnaires pour un ensemble de n ions (e.g. 1073741824 états pour 30 spins $\frac{1}{2}$ — en géneral, pour un système de n spins s, il y a $(2s+1)^n$ états) ; et comme nous ne prenons en compte que les interactions entre spins les plus proches, il y a finalement peu d'éléments non nuls dans la matrice hamiltonienne. Les motivations du physicien pouvant cependant fluctuer tout au long du développement de l'application logicielle, ce travail doit être facilement et rapidement modifiable. Mais ses spécifications peuvent rester informelles : c'est un petit programme qu'il a été demandé de réaliser. Langage prédominant parmi les scientifiques, bénéficiant d'importantes librairies de calcul et suffisamment performant par rapport au C, c'est le FORTRAN qui nous a servi pour écrire ce programme. Il incorpore un type tableau naturellement adapté au calcul vectoriel cher aux physiciens, et il permet aussi, dans sa version 90, une programmation modulaire et meme l'utilisation de certaines techniques inspirées des langages à objets comme l'encapsulation et la surcharge d'opérateurs. Notons donc une forte évolution par rapport au FORTRAN 77 où il fallait encore numéroter les lignes! En outre, la complexité et la taille du problème exige l'utilisation d'ordinateurs à hautes performances et de méthodes de diagonalisation particulières. C'est pourquoi nous serons amenés à faire un peu d'analyse numérique et à exploiter le Cray T3E et son environnement logiciel. La parallélisation d'un programme de simulation — qui existe déjà en version séquentielle — sera donc le thème central de ce stage.

Chapitre 1

Les systèmes de spins

La théorie du magnétisme est étroitement liée aux propriétés des atomes et de leurs électrons. En effet, leur comportement collectif oriente les moments magnétiques individuels. Nous ne considérerons ici que le cas de dimension 1, car c'est le cas le plus étudié et pour lequel nous avons le plus de résultats exacts. D'autre part, certaines propriétés vérifiées en dimension un et deux peuvent permettre de prédire quelles sont les propriétés en dimension trois. Les molécules linéaires, certains cristaux formés de chaînes d'atomes isolées les unes des autres par des ions (comme les cristaux NENP [Mey82]), sont de bons exemples de systèmes magnétiques unidimensionnels.

1.1 Les origines du magnétisme

Certains corps (comme le fer, le nickel, le cobalt ...) peuvent être aimantés spontanément à température ordinaire ; d'un point de vue microscopique, les électrons d'une couche interne incomplète ont en fait leurs spins quasiment alignés dans la même direction. Et, comme chaque moment magnétique individuel est proportionnel à un spin *effectif* [Yos96], cet alignement implique que tous ces moments magnétiques s'ajoutent et forment un aimant macroscopique. Ses propriétés sont déterminées par l'organisation de ces atomes ou ions, qui implique deux effets antagonistes :

- les électrons des couches incomplètes sont délocalisés et sont groupés en bandes;
- ces électrons restent localisés et le magnétisme est dû à chaque atome ou ion.

Généralement, ces deux effets ont lieu simultanément. Cependant, nous allons nous intéresser seulement aux isolants où les porteurs de moments magnétiques sont fixes ; ils sont donc moins difficiles à étudier que les matériaux conducteurs. Et nous nous préoccuperons uniquement des arrangements réguliers et périodiques d'atomes, en champ magnétique extérieur nul.

On dira que le comportement de la chaîne est *ferromagnétique* quand les moments magnétiques ont tendance à être parallèles les uns aux autres, et *antiferromagnétique* quand ils ont tendance à être antiparallèles. En particulier, quand les électrons de valence occupent la couche s (la première couche électronique — le moment orbital étant nul, le spin effectif s'identifie alors au spin réel), on peut montrer que le système est plutôt antiferromagnétique ; au contraire, quand d'autres états de valence sont accessibles (le couplage spin-orbite prédomine sur les interactions entre moments magnétiques individuels), il est plutôt ferromagnétique. Mais lorsqu'on chauffe un matériau ferromagnétique au-dessus d'une température T_c appelée "température de Curie", l'aimantation disparaît et le corps devient *paramagnétique*. Qualitativement, à haute température, l'agitation thermique tend à détruire l'alignement des spins. A température suffisamment basse [Cas89], les isolants peuvent être modélisés par des porteurs de spins effectifs interagissant à courte portée. Nous n'allons donc envisager ici que des interactions entre moments magnétiques voisins

— c'est le modèle de Heisenberg. Dans ce cas, contrairement aux systèmes bi et tridimensionnels, les chaînes magnétiques ne sont ordonnées — des spins plus ou moins distants sont statistiquement corrélés — qu'à l'état fondamental (i.e. l'état de plus basse énergie), tandis que la phase désordonnée est toujours présente à température finie. Avant de commencer l'étude, nous avons besoin de formaliser ce que nous avons dit et d'introduire le vocabulaire de la mécanique quantique.

1.2 Un peu de formalisme quantique

Tout d'abord, on introduit la notion d'opérateur associé à une grandeur physique : l'énergie, par exemple, sera représenté par la matrice hamiltonienne, notée H. Ces opérateurs agissent dans un espace de Hilbert appelé espace des états, car chaque vecteur représente un état du système. Mais l'on ne connaît l'état du système qu'en effectuant une mesure, c'est-à-dire, formellement, en calculant les éléments propres de la matrice associée à la grandeur mesurée. En particulier, si l'on cherche l'énergie du système, on devra la chercher dans le spectre de l'hamiltonien. Cependant, la mécanique quantique a introduit d'autres grandeurs, qui n'ont pas d'analogue classique, comme le *spin*. Pour en savoir plus, on pourra se référer par exemple à [Mes59].

Un spin $\vec{S}$ est un opérateur à trois composantes (chaque composante étant associée à une direction de l'espace), vérifiant les relations de commutation :

$$[S^a, S^b] = i\hbar\varepsilon^{abc}S^c,$$

où ε^{abc} est le tenseur antisymétrique, et où l'on somme sur les indices répétés. Ces composantes sont proportionnelles aux moments magnétiques. Chaque atome présent dans une chaîne étant fixe, nous le repèrerons par son *site*. Tous les sites i de la chaîne ont donc un spin noté $\vec{S}_i$ et carctérisé par le même nombre quantique $s = \frac{1}{2}, 1, \frac{3}{2}, 2, \ldots$ tel que :

$$\vec{S}_i^2 = s(s+1)\hbar^2, \text{ et } S_i^z = m\hbar \text{ où } -s \leq m \leq s.$$

Dans la suite, nous poserons pour simplifier $\hbar \equiv 1$. Pour $s = \frac{1}{2}$, les opérateurs de spin sont représentés par les matrices de Pauli

$$S_i^x = \frac{1}{2}\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad S_i^y = \frac{i}{2}\begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}, \quad S_i^z = \frac{1}{2}\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

qui agissent dans la base canonique :

$$|+\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad |-\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Ces opérateurs n'affectent que le $i^{ème}$ atome. De même, dans la base

$$|+\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}, \quad |0\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}, \quad |-\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix},$$

on aura comme représentation du spin $s = 1$:

$$S_i^+ = \begin{bmatrix} 0 & \sqrt{2} & 0 \\ 0 & 0 & \sqrt{2} \\ 0 & 0 & 0 \end{bmatrix}, \quad S_i^- = \begin{bmatrix} 0 & 0 & 0 \\ \sqrt{2} & 0 & 0 \\ 0 & \sqrt{2} & 0 \end{bmatrix}, \quad S_i^z = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix},$$

où l'on a défini les opérateurs *montant* et *descendant* $S_i^{\pm} = S_i^x \pm iS_i^y$. Ces opérateurs permettent, par une combinaison linéaire des composantes, de calculer simplement l'effet de l'hamiltonien sur

une configuration quelconque de la chaîne. Par exemple, si le $i^{ème}$ spin est dans l'état $|-\rangle$, alors l'action de S_i^+ le transformera en $|+\rangle$ pour des spins $s = \frac{1}{2}$, ou en $|0\rangle$ pour des spins 1.

La limite classique est atteinte pour $\hbar \to 0$ et $s \to \infty$, où les opérateurs de spin commutent et peuvent alors être traités comme les composantes d'un vecteur classique. C'est une bonne approximation pour certains atomes dont le spin est suffisamment grand (e.g. $s = \frac{7}{2}$ pour Gd), alors que d'autres ions sont des systèmes quantiques par excellence (e.g. $s = \frac{1}{2}$ pour Cu, $s = 1$ pour Ni). Tous les systèmes unidimensionnels ferromagnétiques ont sensiblement le même comportement qualitatif, mais, par contre, les systèmes antiferromagnétiques ont des comportements très différents selon la valeur du spin. Pour le modèle d'Heisenberg que nous allons étudier, les énergies du système dépendent beaucoup du caractère entier ou demi-entier du spin, ces dépendances disparaissant à la limite classique. Ainsi, les chaînes de spins [Aff89] peuvent se classer en trois familles: $s = \frac{1}{2}, 1, \infty$.

1.3 Les modèles de chaînes de spins

Faute d'étudier des chaînes infinies en première approximation du système macroscopique — on parlerait alors de limite thermodynamique — on impose des conditions aux limites périodiques, en refermant la chaîne comme sur la figure 1.1. Un état du système est alors un vecteur à n composantes (par exemple, on note $|+--+\rangle$ l'état où les spins 1 et 4 sont dans l'état $|+\rangle$, les spins 2 et 3 dans l'état $|-\rangle$).

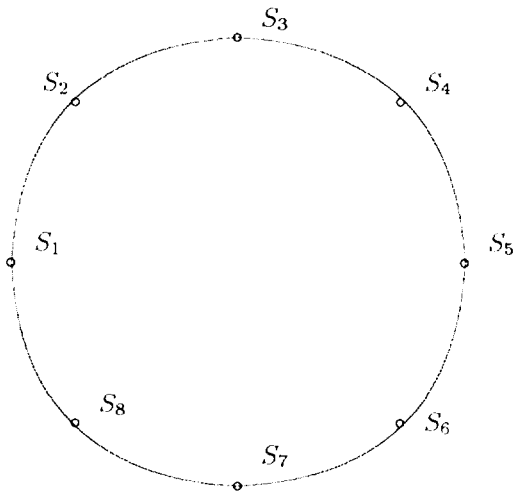

FIG. 1.1 - *Chaîne de spins périodique*

Le modèle de Heisenberg isotrope (i.e. sans direction privilégiée) pour une chaîne ferromagnétique est décrit par l'hamiltonien:

$$H = -J \sum_{i=1}^{n} \vec{S}_i \cdot \vec{S}_{i+1}$$

où $J > 0$ et $\vec{S}_{n+1} \equiv \vec{S}_1$. L'action se décomposera en n termes du type $\vec{S}_i \cdot \vec{S}_{i+1}$ agissant sur la $i^{ème}$

et la $(i+1)^{ème}$ composantes du vecteur d'état. On aura pour des spins $\frac{1}{2}$:

$$\vec{S}_i \cdot \vec{S}_{i+1} |\cdots + + \cdots\rangle = +\frac{1}{4} |\cdots + + \cdots\rangle,$$

$$\vec{S}_i \cdot \vec{S}_{i+1} |\cdots + - \cdots\rangle = -\frac{1}{4} |\cdots + - \cdots\rangle + \frac{1}{2} |\cdots - + \cdots\rangle,$$

$$\vec{S}_i \cdot \vec{S}_{i+1} |\cdots - + \cdots\rangle = -\frac{1}{4} |\cdots - + \cdots\rangle + \frac{1}{2} |\cdots + - \cdots\rangle,$$

$$\vec{S}_i \cdot \vec{S}_{i+1} |\cdots - - \cdots\rangle = +\frac{1}{4} |\cdots - - \cdots\rangle.$$

L'état fondamental est constitué des n spins parallèles $|++\cdots++\rangle$. Les excitations élémentaires sont résolubles exactement pour une déviation de 1 et de 2 spins (quelquesoit la dimension du système). Ainsi la plupart des modèles sont basés sur ces excitations élémentaires appelées *magnons* ; elles permettent une bonne description des propriétés à basse température des systèmes tridimensionnels, mais la formation d'états liés en 1 dimension complique nettement le problème. C'est l'*ansatz* de Bethe [Bet31] qui permit la résolution complète — mais complexe ; c'est pourquoi l'on utilise encore la simulation numérique — des chaînes de spins $s = \frac{1}{2}$; les méthodes "*inverse-scattering*" permettent quant à elles de généraliser les résultats à la limite classique $s \to \infty$. Mais l'ansatz de Bethe ne fonctionne pas pour des chaînes de spin $s \geq 1$; en particulier, Haldane [Hal83] proposa un comportement tout à fait différent pour les chaînes antiferromagnétiques de spin entier. L'hamiltonien d'une chaîne *anti*ferromagnétique s'écrit :

$$H = +J \sum_{i=1}^{n} \vec{S}_i \cdot \vec{S}_{i+1}.$$

L'état fondamental classique — état de Néel — est cette fois constitué des n spins *anti*parallèles comme $|+-+\cdots-+-\rangle$, mais l'état fondamental quantique est une combinaison linéaire d'états stationnaires.

Lorsqu'une direction est privilégiée (par exemple l'axe z), on introduit un coefficient d'anisotropie Δ tel que l'hamiltonien s'écrive :

$$H = \pm J \sum_{i=1}^{n} \vec{S}_i \cdot \vec{S}_{i+1} + (\Delta - 1) S_i^z S_{i+1}^z.$$

Si l'on néglige les autres directions ($\Delta \to \pm\infty$), on parle de modèle d'Ising ; c'est le système le plus simple : les énergies et les états sont directement donnés et sont stationnaires (i.e. n'évoluent pas au cours du temps), mais c'est qualitativement un bon modèle du ferromagnétisme. Cependant, on peut montrer qu'il n'y a pas de transition de phase à 1 dimension, contrairement à ce qui se passe en 2 et 3 dimensions pour le même modèle.

Chapitre 2

Le calcul de valeurs propres

Attardons-nous un peu sur l'algorithme utilisée pour diagonaliser l'hamiltonien, sachant qu'il s'agit d'une matrice symétrique. Une des méthodes les plus anciennes pour calculer la valeur propre dominante est la méthode de la puissance. Généralisée en itération de sous-espace, elle permet de trouver plusieurs valeurs propres dominantes. Enfin, si l'on utilise un ensemble de N méthodes d'itération de N sous-espaces emboités, on parle de méthode QR. La méthode de Lanczos permet, en tridiagonalisant une matrice hermitienne, de calculer ensuite très rapidement les éléments propres par une méthode QR adaptée. Cette dernière méthode autorise une diagonalisation partielle de la matrice : c'est celle que nous allons exploiter ici.

2.1 La méthode de Lanczos

Pour une matrice A, la méthode de la puissance consiste à itérer le produit $x_{k+1} = Ax_k$; on peut montrer que [Las94] :

$$\lim_{k\to\infty} \frac{||x_{k+1}||}{||x_k||} = \lambda$$

où λ est la plus grande valeur propre en module. Pour obtenir les plus petites valeurs propres, on peut appliquer la méthode de la puissance *inverse*, c'est-à-dire l'algorithme précédent à la matrice A^{-1}. Mais cette dernière méthode est surtout utilisée sur la matrice $(A - \mu I)^{-1}$ pour trouver la valeur propre la plus proche de μ et le vecteur propre associé ; cet algorithme a l'avantage de converger rapidement. La méthode de factorisation QR, quant à elle, consiste à appliquer une suite de transformations unitaires : on itère la suite $\{A_k\}$ définie par la récurrence

$$A_0 = A, \quad A_{k+1} = R_k Q_k$$

où Q_k est une matrice unitaire et R_k une matrice triangulaire telles que $A_k \equiv Q_k R_k$. On montre alors que la suite $\{A_k\}$ tend vers une matrice triangulaire, dont la diagonale contient naturellement les valeurs propres.

Cependant, si l'on veut calculer seulement p valeurs propres d'une matrice de taille N et les vecteurs propres correspondants (p étant généralement petit devant N), on utilise la méthode de Lanczos : par projection orthogonale sur un sous-espace, on réduit le problème à la diagonalisation d'une matrice tridiagonale sur ce sous-espace. Soit en effet A une matrice hermitienne ; cette méthode engendre une base orthonormée de vecteurs $\{q_k\}_{1\leq k\leq m}$ d'un sous-espace de dimension m appelé espace de Krylov telle que l'on ait la relation :

$$Aq_k = \beta_{k-1} q_{k-1} + \alpha_k q_k + \beta_k q_{k+1}.$$

La condition d'orthonormalisation est vérifiée si l'on prend [Las94] :

$$\alpha_k \doteq < Aq_k | q_k >, \ \beta_k \doteq < Aq_k | q_{k+1} > .$$

A chaque étape, on calcule donc le vecteur q_{k+1}, que l'on construit explicitement en fonction de q_{k-1}, de q_k, et de l'action de la matrice A sur q_k. L'algorithme se présentera ainsi :

1. On pose $q_0 \leftarrow 0$ et on choisit q_1 tel que $\|q_1\|_2 = 1$.
2. Pour $1 \leq k \leq m$, on calcule :

$$\begin{aligned} r_k &\leftarrow Aq_k - \beta_{k-1} q_{k-1} \\ \alpha_k &\leftarrow < r_k | q_k > \\ r_k &\leftarrow r_k - \alpha_k q_k \\ \beta_k &\leftarrow \|r_k\|_2 \\ q_{k+1} &\leftarrow r_k / \beta_k . \end{aligned}$$

A l'itération m, la projection de l'application linéaire associée à A sur la base $\{q_k\}_{1 \leq k \leq m}$ est une matrice tridiagonale que l'on diagonalise par une méthode QR. Les valeurs propres et vecteurs propres obtenus sont appelés éléments propres de Ritz de la matrice A.

2.2 Utilisation de la méthode

Les valeurs et vecteurs propres de Ritz de A sont des approximations de certains éléments propres de A. L'intérêt de la méthode consiste à garder le nombre d'itérations l petit par rapport à la taille N de la matrice. Or on peut justement montrer [Cha88] que, pour l modeste, les éléments propres de Ritz sont proches des éléments propres de la matrice A associés aux valeurs propres extrêmes, à condition qu'elles soit *assez* distinctes. Par exemple, pour 10 spins $\frac{1}{2}$, soit 1024 états, la plus grande et la plus petite valeur propre sont déterminées après 15 itérations seulement (voir la figure 2.1). De plus, les valeurs propres de la sous-matrice obtenues sont emboîtées d'une itération à l'autre. Mais pour une valeur propre dégénérée, la méthode de Lanczos ne permet pas, en théorie, de trouver tous les vecteurs propres associés ; seules les erreurs d'arrondi permettent la levée complète de dégénérescence si l'on itère suffisamment (en pratique, jusqu'à environ 3 fois la taille de la matrice). Sur la figure 2.1, où l'on a représenté les 3 plus petites et les 3 plus grandes valeurs propres calculées à chaque itération, on voit comment se comporte la méthode : les valeurs proches les unes des autres finissent parfois par se confondre en effectuant des "sauts".

Le principal défaut de cette méthode est qu'elle présente une forte instabilité : le vecteur q_{k+1}, déterminé uniquement par q_{k-1} et par q_k, n'est pas rigoureusement orthogonal aux vecteurs précédents $\{q_j\}_{1 \leq j \leq k-2}$ à cause des erreurs d'arrondi (ainsi que les coefficients α_k et β_k perdent également en précision). On peut pallier ce problème en orthogonalisant les vecteurs $\{q_k\}_{1 \leq k \leq m}$ à chaque itération (méthode de Householder), mais cela requiert la mémorisation prohibitive de tous les vecteurs et un coût de calcul supplémentaire. Une alternative est de contrôler la valeur de β qui, quand elle devient trop petite, est le signe d'une mauvaise orthogonalisation ; dans ce cas, il suffit de réorthogonaliser le vecteur courant q_{k+1} par rapport à certains "bons" vecteurs [GVL83].

2.3 Un peu d'analyse

Soit N la taille de la matrice à diagonaliser ; rappellons qu'elle vaut $(2s+1)^n$ s'il y a n spins s, et que c'est par conséquent une fonction variant exponentiellement par rapport à la taille du problème initial. A la $m^{ème}$ itération, on effectue un produit matrice-vecteur en $O(N^2)$ multiplications pour

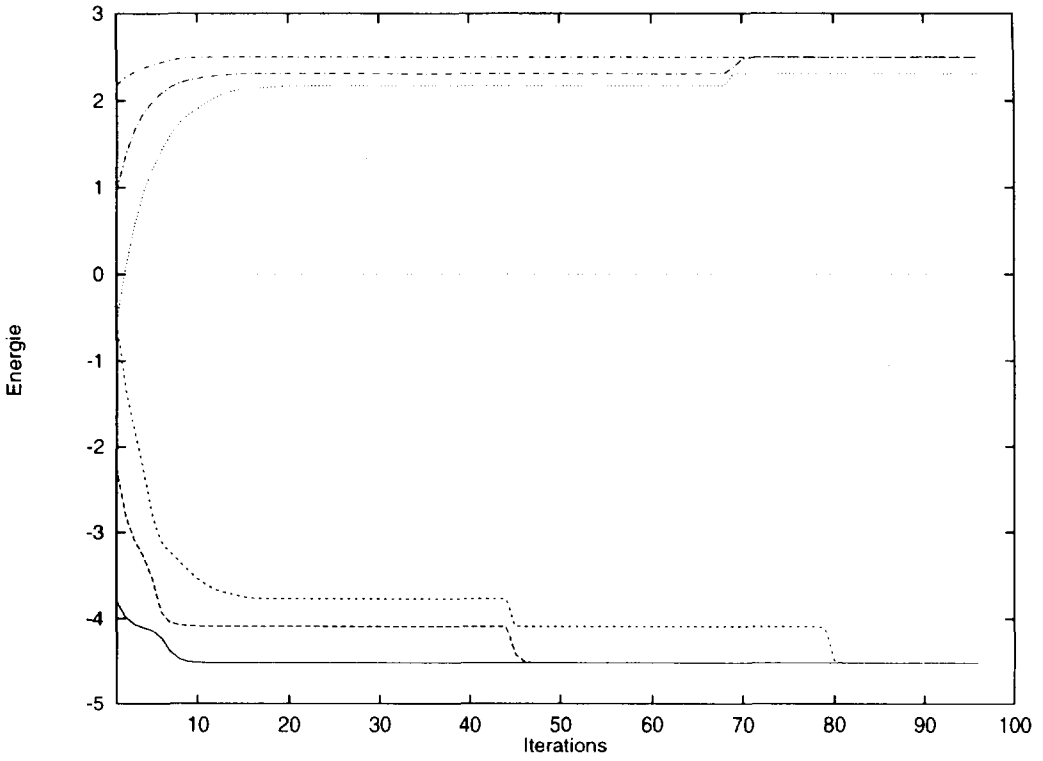

FIG. 2.1 - *Calcul des valeurs extrêmes du spectre pour 10 spins*

tridiagonaliser selon la méthode de Lanczos, et $O(m)$ multiplications pour diagonaliser la sous-matrice tridiagonale par une routine de factorisation QR. Pour l itérations on aura en pratique $l \ll N$ le nombre total de multiplications vaut donc $O(lN^2)$.

Ce qui demande le plus d'opérations arithmétiques est la multiplication matrice-vecteur, en $O(N^2)$ à la chaque itération. La matrice hamiltonienne étant en réalité très creuse, c'est cette multiplication qu'il faudra optimiser et paralléliser en priorité, en considérant cette particularité de la matrice. On pourrait de surcroit envisager d'améliorer de même la phase de diagonalisation QR, mais, utilisée en pratique sur des petites matrices et de plus rapide par rapport au reste du programme (temps linéaire), ce travail ne serait pas très rentable.

Chapitre 3

Le programme

Dans un premier temps, nous avons implémenté la méthode de tridiagonalisation de Lanczos et analysé le comportement global de l'algorithme. Ensuite, nous avons optimisé le produit matrice-vecteur représentant l'action de l'hamiltonien, avant de le paralléliser (voir prochain chapitre). Il existe déjà des programmes de diagonalisation de chaînes de spins, comme «**TITPACK**» du Pr. H. NISHIMORI, mais nous nous intéresserons principalement à la parallélisation d'un programme séquentiel de ce type.

3.1 Présentation générale

Le paramétrage du programme s'effectue interactivement en précisant le nom du fichier (par exemple "`param.txt`") contenant les paramètres du système comme entrée standard lors du lancement (on tapera ainsi la commande «`./spin < param.txt`» si "`spin`" est le nom de l'exécutable). On utilise par conséquent la gestion dynamique des tableaux autorisée par FORTRAN 90 (soit explicitement par la commande `allocate`, soit implicitement par passage de paramètres à une fonction [Lig93]). Les paramètres sont, par ordre d'apparition dans le fichier, la valeur des spins `S`, le nombre de spins `NbSpins`, le coefficient d'anisotropie `Delta` et le nombre d'itérations `NbIterations` de la méthode de Lanczos. Le programme génère ensuite quatre fichiers :

- un fichier "`result.txt`" contenant le spectre calculé à chaque itération ;
- un fichier "`spectre.txt`" contenant le spectre calculé á la dernière itération ;
- un fichier "`bornes.txt`" contenant les bornes du spectre calculé à chaque itération ;
- un fichier "`bornes.gnuplot`" contenant des commandes `gnuplot`.

Le programme «`gnuplot`» est un traceur de courbes et de surfaces ; il nous permet ici de visualiser l'évolution des bornes du spectre calculé à chaque itération. Les informations contenues dans "`result.txt`" sont surtout utiles en phase de test, quand on ignore a priori la durée d'exécution du programme pour des paramètres donnés et que le gestionnaire de batchs l'interrompt avant qu'il n'ai pu se terminer. On s'en passe en phase d'exploitation.

Les différentes routines sont réparties dans trois modules, chaque module étant pour des raisons de maintenabilité dans un fichier différent :

- les routines de diagonalisation dans le module `DIAGONALISATION` ;
- les routines calculant l'action de l'hamiltonien dans le module `PRODUIT` ;
- les routines de lecture/écriture de fichiers dans le module `FILES`.

Nous avons implémenté la méthode de Lanczos dans les routines **diagoDemi** et **diagoGene**, diagonalisant respectivement un hamiltonien d'une chaine de spins $\frac{1}{2}$ et un hamiltonien écrit dans un fichier texte. Ces procédures contiennent également la diagonalisation QR par appel à une routine de la librairie NAG (nommée "F08JEE" sur le Cray ; voir la description en annexe) : cette librairie est une bibliothèque de fonctions et de procédures utiles au calcul scientifique, en général performantes sur des machines séquentielles et vectorielles.

3.2 Version de test

Nous avons d'abord écrit une routine de diagonalisation permettant l'extraction des valeurs propres d'une matrice écrite dans un fichier nommé "**h.txt**". La matrice peut être quelconque à condition d'être symétrique, mais cette méthode présente le double inconvénient d'être pénible à exploiter (on doit écrire tous les éléments de la matrice), et de traiter la matrice comme une matrice pleine alors que l'hamiltonien d'une chaine de spins présente beaucoup de zéros (de la place mémoire est ainsi gaspillée, et des calculs inutiles sont effectués). Les éléments non nuls sont structurés grâce aux symétries du système, mais cette structure est assez difficile à exploiter. Cependant, cette version du programme a permis de valider le bon fonctionnement de l'algorithme de diagonalisation pour des matrices dont on connait le spectre et pour des systèmes encore résolubles à la main, comme les systèmes de deux, trois et quatre spins $\frac{1}{2}$. On a reproduit un exemple d'hamiltonien sur la figure 3.1.

```
1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.5 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.5 0.0 - 1 0.5 0.0 0.0 0.5 0.0 0.5 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.5 0.0
0.0 0.5 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.5 - 1 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.5 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0 0.0 0.0 0.0 0.5 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
```

FIG. 3.1 - *Hamiltonien de Heisenberg d'une chaîne de 4 spins* $\frac{1}{2}$

Une première approche pour traiter une matrice creuse consiste à construire trois vecteurs contenant respectivement les éléments non nuls de la matrice, leurs positions en ligne, et leurs positions en colonne : la procédure s'écrit comme suit :

```
!=============================================================================
SUBROUTINE code(N, H, NbElements, element, row, column)
!=============================================================================
```

```
!! DESCRIPTIF : transforme une matrice creuse en vecteur de ses elements
!!              non nuls et sauvegarde leurs positions
!!
!! DONNEES : 'element' est de dimension 'nbElements', comme 'row' et 'column'

!-------------------------------------------------------------------------------------

  IMPLICIT NONE

  INTEGER, INTENT(in)  :: N, NbElements
  REAL*8,  INTENT(in)  :: H(N,N)
  REAL*8,  INTENT(out) :: element(NbElements)
  INTEGER, INTENT(out) :: row(NbElements), column(NbElements)

  INTEGER ligne, colonne, numero
  REAL valeur

!-------------------------------------------------------------------------------------

  numero = 1

  !! on parcourt la matrice creuse
  DO colonne = 1, N
     DO ligne = 1, N
         valeur = H(ligne, colonne)

         !! on n'enregistre dans le vecteur 'element' que les valeurs non nulles
         !! et dans 'row' et 'column' leurs positions dans la matrice
         IF (valeur /= 0.0) THEN
            element(numero) = valeur
            row(numero) = ligne
            column(numero) = colonne
            numero  = numero  + 1
         END IF
     END DO
  END DO

!=====================================================================================
END SUBROUTINE code
!=====================================================================================
```

Nous avons dû introduire un paramètre supplémentaire, nommé **NbElements**, égal au nombre d'éléments non nuls de la matrice. L'action de l'hamiltonien $v = Hu$ est calculée en effectuant directement le produit matrice-vecteur à partir des vecteurs précédemment construits :

```
!=====================================================================================
SUBROUTINE multGenerale(N, NbElements, element, row, column, u, v)
!=====================================================================================

!! DESCRIPTIF : multiplication matrice-creuse*vecteur : v = Hu ; on a code
!!              la matrice en colonnes dans 'element', contenant 'NbElements'
!!              non nuls positionnes grace a 'row' et 'column'
!!
!! DONNEES : le contenu 'element' de la matrice est de dimension 'NbElements'
!!           'u' et 'v' sont des vecteurs de dimension N
```

```
!-------------------------------------------------------------------------------------

  IMPLICIT NONE

  INTEGER, INTENT(in)  :: N, NbElements
  REAL*8,  INTENT(in)  :: u(N), element(NbElements)
  INTEGER, INTENT(in)  :: row(NbElements), column(NbElements)
  REAL*8,  INTENT(out) :: v(N)

  INTEGER numero, ligne, colonne

!-------------------------------------------------------------------------------------

  !! initialisation du vecteur resultat
  v(:) = 0.0

  !! on n'opere la multiplication que sur les elements non nuls
  DO numero = 1, NbElements
     ligne = row(numero)
     colonne = column(numero)
     v(ligne) = v(ligne) + element(numero) * u(colonne)
  END DO

!=====================================================================================
END SUBROUTINE multGenerale
!=====================================================================================
```

Comme nous l'avons dit, ces routines ont permis de tester l'implémentation de la méthode de Lanczos, qui nous a donné des résultats satisfaisants. Cependant, l'intérêt de cette simulation s'arrete là si l'on ne peut pas calculer automatiquement l'hamiltonien d'une chaine de longueur quelconque. C'est pourquoi nous sommes amenés à écrire des procédures spécifiques : elles calculeront non pas l'hamiltonien lui-même, mais son action sur un vecteur quelconque. En effet, nous avons vu que la manipulation de la matrice hamiltonienne comme un tableau bidimensionnel est lourde puisqu'elle est énorme et très creuse.

3.3 Version d'exploitation

Nous sommes ainsi amenés à perdre la généralité d'un programme qui aurait implémenté un type abstrait SPIN, en écrivant de manière spécifique un algorithme de calcul de l'action d'un hamiltonien d'une chaine de spins de valeur s donnée. Nous avons commencé par le plus facile, c'est-à-dire par des spins $\frac{1}{2}$. Les équations (voir la section 1.3) donnant l'action de $\vec{S}_i \cdot \vec{S}_{i+1}$ sur la $i^{ème}$ et la $(i+1)^{ème}$ composantes du vecteur d'état étant valables quelquesoient les valeurs des autres composantes, nous devons boucler ce calcul sur toutes ces valeurs possibles. Il s'agit en fait de numéroter — en développement binaire: $|--\cdots-\rangle \equiv 0, |--\cdots+\rangle \equiv 1, \ldots$ — tous les 2^n états accessibles, et d'appliquer les quatre formules correspondant aux quatres configurations du couple de spins $(i, i+1)$ au vecteur sur lequel s'exerce l'action de l'hamiltonien. Puis il faut sommer les contributions des n spins en prenant garde à la périodicité de la chaine. Notons que nous sommes alors restreints à des chaines de spins de longeur supérieure ou égale à 3 pour le bon fonctionnement de la boucle. Voici le code d'une telle procédure de calcul :

```
!=====================================================================================
SUBROUTINE multDemi(NbSpins, N,  u, v)
!=====================================================================================
```

```
!! DESCRIPTIF : on construit l'action de l'hamiltonien de Heisenberg pour S=1/2

!-------------------------------------------------------------------------------

  IMPLICIT NONE

  REAL*8, PARAMETER :: DEMI = 0.5, QUART = 0.25

  INTEGER, INTENT(in)  :: N, NbSpins
  REAL*8,  INTENT(in)  :: u(N)
  REAL*8,  INTENT(out) :: v(N)

  INTEGER, DIMENSION(0:NbSpins+1) :: p2

  INTEGER i, j, poidsFaibles, poidsForts, spin, s0, s1, k

!-------------------------------------------------------------------------------

  v(:) = 0.0
  DO spin = 0, NbSpins+1
     p2(spin) = 2**spin
  END DO

  DO spin = 1, NbSpins-1
     DO k = 0, 3
        s0 = MOD(k,2)
        s1 = k/2
        DO poidsForts = 0, p2(NbSpins-spin-1)-1
           DO poidsFaibles = 0, p2(spin-1)-1
              i = poidsForts * p2(spin+1) + poidsFaibles &
                  + s0 * p2(spin-1) + s1 * p2(spin) + 1
              j = poidsForts * p2(spin+1) + poidsFaibles &
                  + s1 * p2(spin-1) + s0 * p2(spin) + 1
              v(i) = v(i) - QUART * u(i) + DEMI * u(j)
           END DO
        END DO
     END DO
  END DO

  DO k = 0, 3
     s0 = MOD(k,2)
     s1 = k/2
     DO poidsForts = 0, p2(NbSpins-2)-1
        i = poidsForts * 2 + s0 + s1 * p2(NbSpins-1) + 1
        j = poidsForts * 2 + s1 + s0 * p2(NbSpins-1) + 1
        v(i) = v(i) - QUART * u(i) + DEMI * u(j)
     END DO
  END DO

!===============================================================================
END SUBROUTINE multDemi
!===============================================================================
```

Pour éviter de répéter les calculs de puissances de 2, nous les avons mémorisées une fois pour toutes dans le vecteur `p2`. Puisqu'il faut compter en binaire les états accessibles aux spins à gauche (notés `poidsForts`) et à droite (notés `poidsFaibles`), nous avons par la suite utilisé des fonctions prédéfinies du FORTRAN qui manipulent les bits de nombres entiers pour effectuer les boucles.

Ces fonctions [Lig93] ont l'avantage d'expliciter davantage le dénombrement en base 2 et surtout d'être extrêment rapides à l'exécution (le temps est diminué d'environ 25%). Nous avons du alors introduire une variable **masque** pour pouvoir insérer les quatre configurations du couple $(i, i+1)$ dans le décompte total. La routine prend cette forme :

```
!=============================================================================
SUBROUTINE multDemi(NbSpins, N,  Delta, u, v)
!=============================================================================

!! DESCRIPTIF : on construit l'action de l'hamiltonien de Heisenberg pour S=1/2

!-----------------------------------------------------------------------------

  IMPLICIT NONE

  REAL*8, PARAMETER :: DEMI = 0.5, QUART = 0.25

  INTEGER, INTENT(in)  :: N, NbSpins
  REAL*8,  INTENT(in)  :: u(N), Delta
  REAL*8,  INTENT(out) :: v(N)

  INTEGER i, j, spin, etat, masque, etatPartiel

!-----------------------------------------------------------------------------

  v(:) = 0.0

  DO etatPartiel = 0, N/4 - 1

     !! action des Si.Si+1
     DO spin = 0, NbSpins-2
        masque = IBSET(0, spin) - 1
        etat = ISHFT(IAND(NOT(masque), etatPartiel), 2) + IAND(masque, etatPartiel)

        !! etat --
        i = IBCLR(IBCLR(etat, spin), spin + 1) + 1
        v(i) = v(i) + QUART * Delta * u(i)

        !! etat -+
        i = IBCLR(IBSET(etat, spin), spin + 1) + 1
        j = IBSET(IBCLR(etat, spin), spin + 1) + 1
        v(i) = v(i) - QUART * u(i) + DEMI * u(j)

        !! etat +-
        i = IBSET(IBCLR(etat, spin), spin + 1) + 1
        j = IBCLR(IBSET(etat, spin), spin + 1) + 1
        v(i) = v(i) - QUART * u(i) + DEMI * u(j)

        !! etat ++
        i = IBSET(IBSET(etat, spin), spin + 1) + 1
        v(i) = v(i) + QUART * Delta * u(i)

     END DO

     !! action de Sn.S1
     etat = ISHFT(etatPartiel, 1)
```

```
      !! etat --
      i = IBCLR(IBCLR(etat, NbSpins-1), 0) + 1
      v(i) = v(i) + QUART * Delta * u(i)

      !! etat -+
      i = IBCLR(IBSET(etat, NbSpins-1), 0) + 1
      j = IBSET(IBCLR(etat, NbSpins-1), 0) + 1
      v(i) = v(i) - QUART * u(i) + DEMI * u(j)

      !! etat +-
      i = IBSET(IBCLR(etat, NbSpins-1), 0) + 1
      j = IBCLR(IBSET(etat, NbSpins-1), 0) + 1
      v(i) = v(i) - QUART * u(i) + DEMI * u(j)

      !! etat ++
      i = IBSET(IBSET(etat, NbSpins-1), 0) + 1
      v(i) = v(i) + QUART * Delta * u(i)

    END DO

!=============================================================================================
END SUBROUTINE multDemi
!=============================================================================================
```

Notons que si l'on veut simuler le modèle d'Ising, on donnera un coefficient d'anisotropie très élevé, mais il faudra alors normaliser les énergies obtenues après diagonalisation en divisant par ce même coefficient pour avoir les vraies valeurs. Cette dernière version présente l'inconvénient d'être spécifiquement dédiée aux spins $\frac{1}{2}$; la version précédente est plus facilement adaptable : pour des spins 1 par exemple, il suffit de remplacer le tableau `p2` par un tableau qui contient les puissances de 3, et de changer les formules exprimant l'action de $\vec{S}_i \cdot \vec{S}_{i+1}$. Le comptage en ternaire est cependant moins trivial et il n'existe pas de fonctions prédéfinies correspondantes. Il resterait donc à écrire une routine de calcul d'hamiltonien pour des chaines de spins $s \geq 1$ [Gol92]. Mais il y a d'autres limitations importantes comme la taille des données ; à partir de 21 spins, les stations Sun comme les calculateurs Cray ne disposent pas d'assez de mémoire pour effectuer le calcul. Celui-ci devient en outre rapidement très long, même si la complexité du produit matrice-vecteur a été, par rapport à celle du produit brut, ramenée de $O(N^2)$ à $O(nN)$ où $N \equiv 2^n$. C'est pourquoi nous allons adapter ce programme à l'architecture parallèle du Cray T3E.

Chapitre 4

La parallélisation

Nous allons donc maintenant nous attaquer à un problème nouveau : la parallélisation des routines que nous avons d'abord écrites de façon séquentielle. Nous avons déjà vu quel partie du programme nécessite le plus d'être optimisée ; c'est l'action de l'hamiltonien, dont la taille varie exponentiellement par rapport à la dimension du système physique. Nous disposons de deux ordinateurs à hautes performances : un Cray C94 vectoriel et un Cray T3E parallèle, mais nous ne développerons spécifiquement que pour le deuxième (on peut toujours exécuter la version séquentielle du programme sur le C94, dont le compilateur s'efforcera de vectoriser les boucles).

4.1 Environnement

Le Cray T3E du CEA que nous exploitons est un calculateur à 304 processeurs DEC Alpha EV5.6 cadencés à 375 MHz ayant chacun entre 128 Mo et 1Go de mémoire, dont 6 sont réservés à la gestion du système. L'architecture est donc distribuée, et le parallélisme des programmes est basé sur l'envoi de messages. Le réseau d'interconnexion est un tore 3D virtuellement totalement connecté. Le système d'exploitation est UNICOS/MK (UNIX développé pour les ordinateurs Cray) et permet le lancement de processus de deux manières :

- soit par commande interactive à travers le shell ;
- soit par soumission de batch interprété par un gestionnaire NQS.

Il y a plusieurs protocoles d'envois de message disponibles : PVM, SHMEM, MPI. Pour sa simplicité, nous avons choisi SHMEM, dont nous avons reproduit le «`man`» en annexe.

Disposant d'un temps CPU limité en interactif (afin d'optimiser le partage des ressources du calculateur entres plusieurs personnes), nous sommes contraints de lancer des batchs NQS dés que le programme est un peu long à s'exécuter, soit en phase d'exploitation. Ceux-ci se présentent comme une succession de déclarations — commande «`# QSUB`» — liées au déroulement du processus que l'on lance (priorité, nombre de processeurs, temps maximal ...). Puis l'on ajoute des commandes UNIX pour lancer la compilation et l'exécution avec les bons paramètres (voir la figure 4.1). On utilise les options de compilation suivantes :

- l'option `-X` précise le nombre de processeurs sur lesquels on désire travailler ;
- les options `-eA -lapp` permettent l'utilisation du programme «`apprentice`» qui mesure les performances du programme parallèle ;
- l'option `-l ~/nag.a` fait le lien avec la librairie NAG.

```
# QSUB -r spin                                  # nom du job
# QSUB -q jour                                  # tarif jour/nuit/wend
# QSUB -l mpp_p=12                              # nb de PE
# QSUB -l mpp_t=0:45:00                         # temps limite hh:mm:ss en parallele
# QSUB -lT 0:45:00 -lt 0:40:00 -eo              # temps limite hh:mm:ss en sequentiel
# QSUB -me -mu lamarcq@wasa.saclay.cea.fr       # e-mail a la fin du job

set npes = '/usr/bin/limit | sed -n 5p'         # recuperation nb de PE
set echo

cd /u/lamarcq/parallele/spins/                  # repertoire de travail
mpprun -n $npes ./spin < param$npes             # lancement sur npes processeurs
```

FIG. 4.1 - *Exemple de batch NQS*

La variable **npes** indique le nombre de processeurs utilisés (ici 12) pour exécuter le programme. Cependant, avant de pouvoir lancer le programme par l'instruction «**mpprun**», il faut avoir compilé auparavant avec l'option de compilation "**-X m**". Nous avons choisi par ailleurs comme convention d'appeler "**paramP**" le fichier contenant les paramètres du système s'il requiert un calcul sur P processeurs.

4.2 Première parallélisation

Dans un premier temps, nous avons amélioré la vitesse du programme. La parallélisation la plus simple consiste à décomposer l'action de l'hamiltonien en n actions simultanées calculées séparément : l'opérateur du type $\vec{S}_i \cdot \vec{S}_{i+1}$ sera calculé sur le processeur P_i (à chaque spin correspondra ainsi un processeur). Pour obtenir le bon résultat final, il faut ensuite sommer chaque contribution, c'est-à-dire effectuer une *réduction* de l'ensemble des vecteurs calculés localement. Les communications entre processeurs apportent de nouvelles difficultés, d'autant plus que les documentations spécifiant les primitives utilisées sont parfois elliptiques. En effet, nous avons été confrontés à un problème lors de l'utilisation de la procédure **SHMEM_REAL8_SUM_TO_ALL** : cette primitive de la librairie SHMEM exige l'initialisation de variables de synchronisation (**pSync** et **pWrk**) et d'une barriere de synchronisation (**SHMEM_BARRIER_ALL**), mais nécessite en plus l'appel à une directive de compilation — appelée «**!DIR$ SYMMETRIC**» — pour déclarer les arguments échangés (variable **v**) à la meme adresse locale sur tous les processeurs. Faute de les déclarer ainsi, le programme ne se comporte plus de manière déterministe et il semble exister une limite dans la taille (environ 256 réels de 64 bits) de la variable **v**.

Sur la figure 4.2, nous avons comparé, pour des systèmes de taille modeste, les performances du programme séquentiel et celles du programme parallèle s'exécutant sur le Cray T3E. Nous voyons que le gain est assez médiocre, surtout parce que le nombre de processeurs utilisés croit avec la taille du problèm. Mais nous retrouvons tout de même le fait qu'il faut privilégier le traitement de grands problèmes sur les ordinateurs parallèles. L'analyse par le programme «**apprentice**» nous donne le temps passsé dans chaque routine avec une précision quelque peu arbitraire en pratique, mais qui permet tout de meme de comparer les vitesses relatives des différentes parties du programme. Ainsi, la majeure partie du temps de calcul est passée dans la procédure **multDemi** calculant l'action de

l'hamiltonien, et les communications sont encore très minoritaires. La principale limitation, comme nous l'avons déjà évoquée à la section 3.3, est la place mémoire disponible : on ne peut pas calculer des systèmes de plus de 21 spins ! Nous avons donc envisagé une autre méthode de parallélisation qui distribue les données sur les mémoires locales des processeurs.

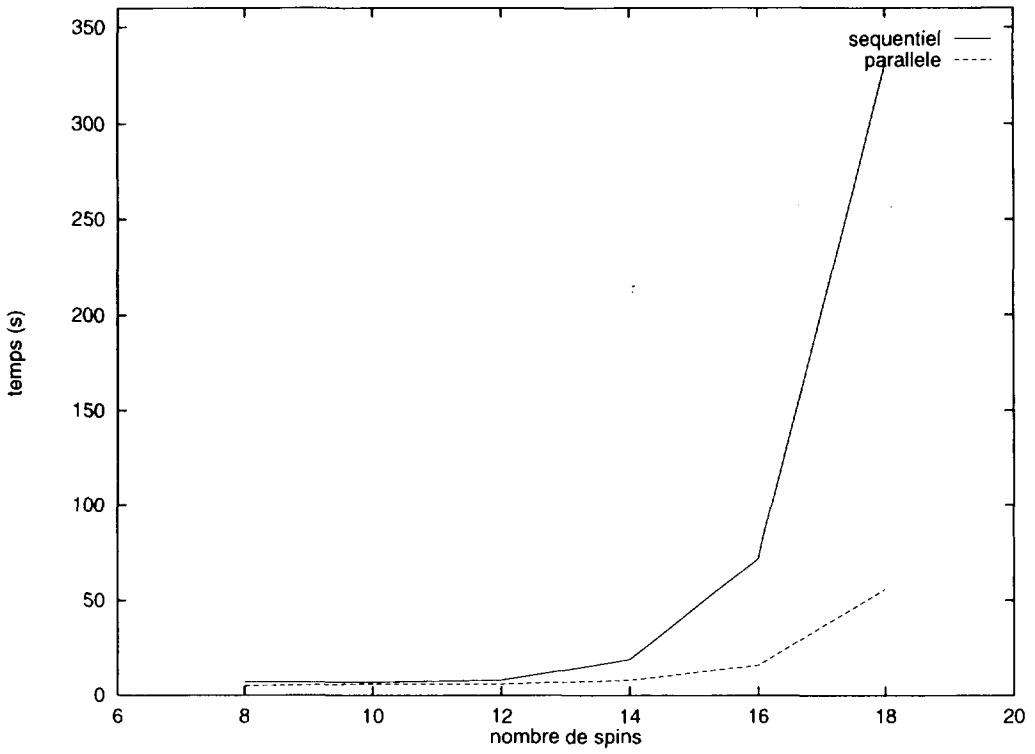

FIG. 4.2 - *Performances des programmes séquentiel et parallèle (n processeurs) sur Cray T3E*

4.3 Seconde parallélisation

Les données critiques sont les vecteurs d'état, puisqu'ils sont de longueur $N = 2^n$, soit d'une taille de 2 méga mots de 64 bits (réels double précision) s'il y a 21 spins. Il faut donc les diviser afin d'en répartir les composantes sur plusieurs processeurs. Pour cela, on peut associer un sous-ensemble d'états à chaque processeur. Par exemple, on dispose les états pairs sur un premier processeur, et les états impairs sur un deuxième processeur, ce qui permet d'augmenter la taille du système par deux (donc de rajouter un spin). Cette méthode est toujours dépendante du nombre de spins du système étudié (cette fois-ci, le nombre de processeurs requis croît exponentiellement en 2^{n-21} pour n spins — en fonction de la taille du problème). Mais surtout, elle complique l'algorithme fortement, puisqu'il faut effectuer des communications entre processeurs lorsqu'il y a des interactions entre états situés sur différents processeurs. En outre, il est nécessaire d'implémenter

une fonction de produit scalaire parallèle puisque chaque processeur diagonalise désormais une partie seulement de la matrice hamiltonienne.

Nous avons donc éclaté explicitement les vecteurs de base sur quatre processeurs (pour commencer), dont les numéros — 00, 01, 10, et 11 — correspondent aux deux poids les plus forts du vecteur d'état. Nous avons dû considérer trois cas par cette méthode :

- la somme $\sum_{i=1}^{n-2} \vec{S}_i \cdot \vec{S}_{i+1}$ est calculée comme dans le cas séquentiel sur chaque processeur, en bouclant le calcul sur le spin i et sur les états de base du sous-ensemble de $n-2$ spins ;
- les produits $\vec{S}_{n-2} \cdot \vec{S}_{n-1}$ et $\vec{S}_n \cdot \vec{S}_1$ sont calculés en cherchant les états correspondant aux spins $n-1$ et n sur les processeurs concernés, selon la valeur de ces spins ;
- le produit $\vec{S}_{n-1} \cdot \vec{S}_n$ est calculé simplement en considérant chaque processeur (donc chaque configuration de spins) séparément.

Comme prévu, nous pouvons ainsi traiter des problèmes à 23 spins. Si l'on ne peut réduire la complexité exponentielle du problème, le gain s'améliore par contre notablement (voir figure 4.3). En effet, les calculs sont répartis sur les quatre processeurs et les communications sont plus rapides que la réduction effectuée dans la première version parallèle du programme.

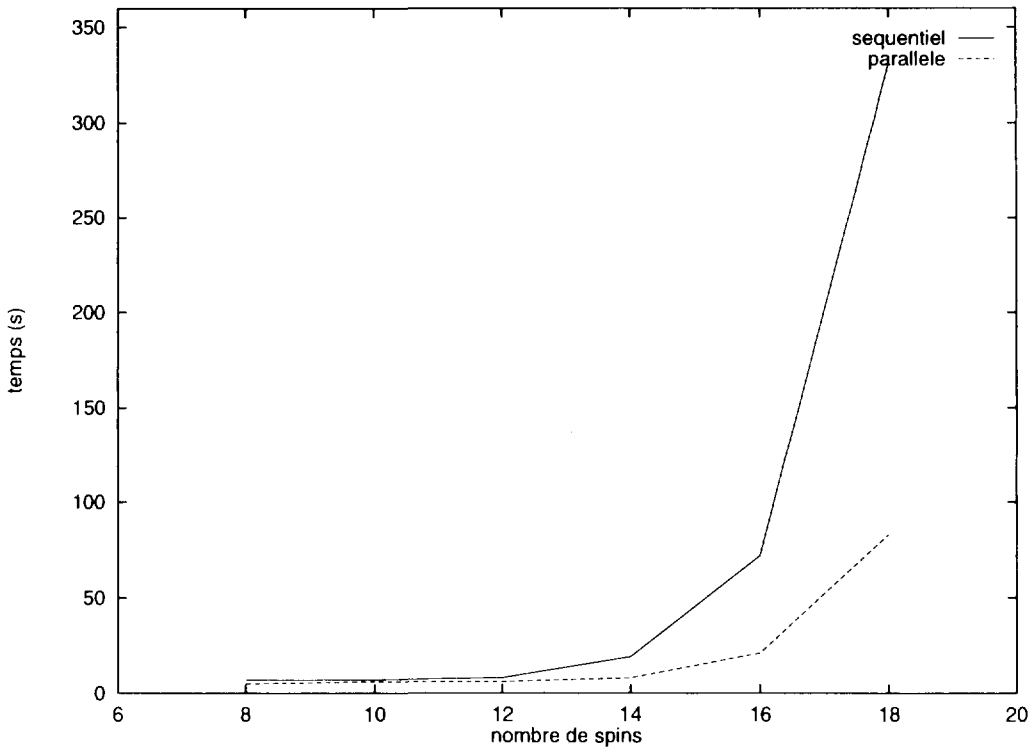

FIG. 4.3 - *Performances des programmes séquentiel et parallèle (4 processeurs) sur Cray T3E*

Conclusion

Ce stage a été l'occasion d'apprendre beaucoup de choses, outre l'apprentissage technique (utilisation du FORTRAN, programmation sur machine parallèle à architecture distribuée) ; en effet, nous avons dû nous investir dans la compréhension de certains modèles physiques et de méthodes particulières d'analyse numérique. Mais le principal est sans doute l'intégration d'un laboratoire de recherche fondamentale. Le travail que nous avons fait est seulement préparatoire. Si les simulations que nous avons lancées n'ont rien appris sur la physique, il faudra étendre la simulation réalisée à des systèmes moins connus, comme des systèmes de spins à interactions quelconques, ou dont le spin est supérieur ou égal à 1. L'intérêt de notre programme réside donc dans la parallélisation du calcul de chaînes de spins $\frac{1}{2}$.

Bibliographie

[Aff89] I. Affleck, *Quantum spin chains and the Haldane gap*, J. Phys.: Cond. Matter **1**, 3047-3072 (1989)

[Bet31] H. Bethe, *Eigenvalues and Eigenfunctions of a linear chain of atoms*, Z. Physik **71**, 205-226 (1931)

[Cas89] W.J. Caspers, *Spin systems*, World Scientific (1989)

[Cha88] F. Chatelin, *Valeurs propres de matrices*, Masson (1988)

[Bon64] J.C. Bonner and M.E. Fisher, *Linear magnetic chains with anisotropic coupling*, Phys. Rev. A **135**, 640-658 (1964)

[Gol92] O. Golinelli, Th. Jolicoeur, and R. Lacaze, *Haldane gaps in a spin-1 Heisenberg chain with easy-plane single-ion anisotropy*, Phys. Rev. B **45**, 9798-9805 (1992)

[GVL83] G.H. Golub and C.F. Van Loan, *Matrix Computations*, The John Hopkins University Press (1983)

[Hal83] F.D.M. Haldane, *Nonlinear field theory of large-spin Heisenberg antiferromagnets: semiclassically quantized solitons of the one-dimensionnal easy-axis Néel state*, Phys. Rev. Lett. **50**, 1153-1156 (1983)

[Las94] P. Lascaux, R. Théodor, *Analyse numérique matricielle appliquée à l'art de l'ingénieur*, Masson (1994)

[Lig93] P. Lignelet, *FORTRAN 90 - Approche par la pratique*, Série Informatique (1993)

[Mes59] A. Messiah, *Mécanique quantique*, Dunod (1959)

[Mey82] A. Meyer, A. Gleizes, J.-J. Girerd, M. Verdaguer, and O. Khan, *Crystal structures, magnetic anisotropy properties, and orbital interactions in catena-(μ-nitrito)-bis(ethylenediamine)nickel(II) perchlorate and triiodide*, Inorg. Chem. **21**, 1729-1739 (1982)

[Ren88] J.P. Renard, M. Verdaguer, L.P. Regnault, W.A.C. Erkelens, J. Rossat-Mignod, and W.G. Stirling, *Presumption for a quantum energy gap in the quasi-one-dimensionnal $S = 1$ Heisenberg antiferromagnet $Ni(C_2H_8N_2)_2NO_2(ClO_4)$*, Europhys. Lett. **3** (8), 945-951 (1987)

[Yos96] K. Yosida, *Theory of magnetism*, Springer (1996)

Annexe A

La librairie NAG

F08JEF (SSTEQR/DSTEQR) – NAG Fortran Library Routine Document

Note: before using this routine, please read the Users' Note for your implementation to check the interpretation of *bold italicised* terms and other implementation-dependent details. The routine name may be precision-dependent.

1. Purpose

F08JEF (SSTEQR/DSTEQR) computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric tridiagonal matrix, or of a real symmetric matrix which has been reduced to tridiagonal form.

2. Specification

```
SUBROUTINE F08JEF (COMPZ, N, D, E, Z, LDZ, WORK, INFO)
ENTRY      ssteqr  (COMPZ, N, D, E, Z, LDZ, WORK, INFO)
INTEGER            N, LDZ, INFO
real               D(*), E(*), Z(LDZ,*), WORK(*)
CHARACTER*1        COMPZ
```

The ENTRY statement enables the routine to be called by its LAPACK name.

3. Description

This routine computes all the eigenvalues, and optionally all the eigenvectors, of a real symmetric tridiagonal matrix T. In other words, it can compute the spectral factorization of T as

$$T = Z\Lambda Z^T,$$

where Λ is a diagonal matrix whose diagonal elements are the eigenvalues λ_i, and Z is the orthogonal matrix whose columns are the eigenvectors z_i. Thus

$$Tz_i = \lambda_i z_i \text{ for } i = 1,2,...,n.$$

The routine may also be used to compute all the eigenvalues and eigenvectors of a real symmetric matrix A which has been reduced to tridiagonal form T:

$$A = QTQ^T, \text{ where } Q \text{ is orthogonal,}$$
$$= (QZ)\Lambda(QZ)^T.$$

In this case, the matrix Q must be formed explicitly and passed to F08JEF, which must be called with COMPZ = 'V'. The routines which must be called to perform the reduction to tridiagonal form and form Q are:

full matrix	F08FEF (SSYTRD/DSYTRD) + F08FFF (SORGTR/DORGTR)
full matrix, packed storage	F08GEF (SSPTRD/DSPTRD) + F08GFF (SOPGTR/DOPGTR)
band matrix	F08HEF (SSBTRD/DSBTRD) with VECT = 'V'.

F08JEF uses the implicitly shifted QR algorithm, switching between the QR and QL variants in order to handle graded matrices effectively (see Greenbaum and Dongarra [2]). The eigenvectors are normalized so that $\|z_i\|_2 = 1$, but are determined only to within a factor ± 1.

If only the eigenvalues of T are required, it is more efficient to call F08JFF (SSTERF/DSTERF) instead. If T is positive-definite, small eigenvalues can be computed more accurately by F08JGF (SPTEQR/DPTEQR).

4. References

[1] GOLUB, G.H. and VAN LOAN, C.F.
Matrix Computations, §8.2.
Johns Hopkins University Press, Baltimore, Maryland, (2nd Edition) 1989.

[2] GREENBAUM, A. and DONGARRA, J.J.
Experiments with QR/QL Methods for the Symmetric Tridiagonal Eigenproblem.
LAPACK Working Note No. 17 (Technical Report CS-89-92), University of Tennessee, Knoxville, 1989.

[3] PARLETT, B.N.
The Symmetric Eigenvalue Problem, §8.
Prentice-Hall, Englewood Cliffs, New Jersey, 1980.

5. Parameters

1: COMPZ – CHARACTER*1. *Input*

On entry: indicates whether the eigenvectors are to be computed as follows:

if COMPZ = 'N', then only the eigenvalues are computed (and the array Z is not referenced);

if COMPZ = 'T', then the eigenvalues and eigenvectors of T are computed (and the array Z is initialized by the routine);

if COMPZ = 'V', then the eigenvalues and eigenvectors of A are computed (and the array Z must contain the matrix Q on entry).

Constraint: COMPZ = 'N', 'T' or 'V'.

2: N – INTEGER. *Input*

On entry: n, the order of the matrix T.

Constraint: N ≥ 0.

3: D(*) – *real* array. *Input/Output*

Note: the dimension of the array D must be at least max(1,N).

On entry: the diagonal elements of the tridiagonal matrix T.

On exit: the n eigenvalues in ascending order, unless INFO > 0 (in which case see Section 6).

4: E(*) – *real* array. *Input/Output*

Note: the dimension of the array E must be at least max(1,N–1).

On entry: the off-diagonal elements of the tridiagonal matrix T.

On exit: the array is overwritten.

5: Z(LDZ,*) – *real* array. *Input/Output*

Note: the second dimension of the array Z must be at least max(1,N) if COMPZ = 'V' or 'T', and at least 1 if COMPZ = 'N'.

On entry: if COMPZ = 'V', Z must contain the orthogonal matrix Q from the reduction to tridiagonal form. If COMPZ = 'T', Z need not be set.

On exit: if COMPZ = 'T' or 'V', the n required orthonormal eigenvectors stored by columns; the ith column corresponds to the ith eigenvalue, where $i = 1,2,...,n$, unless INFO > 0.

Z is not referenced if COMPZ = 'N'.

6: LDZ – INTEGER. *Input*

On entry: the first dimension of the array Z as declared in the (sub)program from which F08JEF (SSTEQR/DSTEQR) is called.

Constraints: LDZ ≥ 1 if COMPZ = 'N',
LDZ ≥ max(1,N) if COMPZ = 'V' or 'T'.

7: WORK(*) – *real* array. *Workspace*

Note: the dimension of the array WORK must be at least max(1,2*(N–1)) if COMPZ = 'V' or 'T', and at least 1 if COMPZ = 'N'.

WORK is not referenced if COMPZ = 'N'.

8: INFO – INTEGER. *Output*

On exit: INFO = 0 unless the routine detects an error (see Section 6).

6. Error Indicators and Warnings

INFO < 0

If INFO = $-i$, the ith parameter had an illegal value. An explanatory message is output, and execution of the program is terminated.

INFO > 0

The algorithm has failed to find all the eigenvalues after a total of 30×N iterations. In this case, D and E contain on exit the diagonal and off-diagonal elements, respectively, of a tridiagonal matrix orthogonally similar to T. If INFO = i, then i off-diagonal elements have not converged to zero.

7. Accuracy

The computed eigenvalues and eigenvectors are exact for a nearby matrix $T + E$, where

$$\|E\|_2 = O(\varepsilon)\|T\|_2,$$

and ε is the ***machine precision***.

If λ_i is an exact eigenvalue and $\tilde{\lambda}_i$ is the corresponding computed value, then

$$|\tilde{\lambda}_i - \lambda_i| \leq c(n)\varepsilon\|T\|_2,$$

where $c(n)$ is a modestly increasing function of n.

If z_i is the corresponding exact eigenvector, and $\tilde{z}_i$ is the corresponding computed eigenvector, then the angle $\theta(\tilde{z}_i, z_i)$ between them is bounded as follows:

$$\theta(\tilde{z}_i, z_i) \leq \frac{c(n)\varepsilon\|T\|_2}{\min_{i \neq j}|\lambda_i - \lambda_j|}.$$

Thus the accuracy of a computed eigenvector depends on the gap between its eigenvalue and all the other eigenvalues.

8. Further Comments

The total number of floating-point operations is typically about $24n^2$ if COMPZ = 'N' and about $7n^3$ if COMPZ = 'V' or 'T', but depends on how rapidly the algorithm converges. When COMPZ = 'N', the operations are all performed in scalar mode; the additional operations to compute the eigenvectors when COMPZ = 'V' or 'T' can be vectorized and on some machines may be performed much faster.

The complex analogue of this routine is F08JSF (CSTEQR/ZSTEQR).

9. Example

To compute all the eigenvalues and eigenvectors of the symmetric tridiagonal matrix T, where

$$T = \begin{pmatrix} -6.99 & -0.44 & 0.00 & 0.00 \\ -0.44 & 7.92 & -2.63 & 0.00 \\ 0.00 & -2.63 & 2.34 & -1.18 \\ 0.00 & 0.00 & -1.18 & 0.32 \end{pmatrix}.$$

See also the examples for F08FFF, F08GFF or F08HEF, which illustrate the use of this routine to compute the eigenvalues and eigenvectors of a full or band symmetric matrix.

9.1. Program Text

Note: the listing of the example program presented below uses *bold italicised* terms to denote precision-dependent details. Please read the Users' Note for your implementation to check the interpretation of these terms. As explained in the Essential Introduction to this manual, the results produced may not be identical for all implementations.

```
*     F08JEF Example Program Text
*     Mark 16 Release. NAG Copyright 1992.
*     .. Parameters ..
      INTEGER          NIN, NOUT
      PARAMETER        (NIN=5,NOUT=6)
      INTEGER          NMAX, LDZ
      PARAMETER        (NMAX=8,LDZ=NMAX)
*     .. Local Scalars ..
      INTEGER          I, IFAIL, INFO, N
*     .. Local Arrays ..
      real             D(NMAX), E(NMAX-1), WORK(2*NMAX-2), Z(LDZ, NMAX)
*     .. External Subroutines ..
      EXTERNAL         ssteqr, X04CAF
*     .. Executable Statements ..
      WRITE (NOUT,*) 'F08JEF Example Program Results'
*     Skip heading in data file
      READ (NIN,*)
      READ (NIN,*) N
      IF (N.LE.NMAX) THEN
*
*        Read T from data file
*
         READ (NIN,*) (D(I),I=1,N)
         READ (NIN,*) (E(I),I=1,N-1)
*
*        Calculate all the eigenvalues and eigenvectors of T
*
         CALL ssteqr('I',N,D,E,Z,LDZ,WORK,INFO)
*
         WRITE (NOUT,*)
         IF (INFO.GT.0) THEN
            WRITE (NOUT,*) 'Failure to converge.'
         ELSE
*
*           Print eigenvalues and eigenvectors
*
            WRITE (NOUT,*) 'Eigenvalues'
            WRITE (NOUT,99999) (D(I),I=1,N)
            WRITE (NOUT,*)
            IFAIL = 0
*
            CALL X04CAF('General',' ',N,N,Z,LDZ,'Eigenvectors',IFAIL)
*
         END IF
      END IF
      STOP
*
99999 FORMAT (3X,(8F8.4))
      END
```

9.2. Program Data

```
F08JEF Example Program Data
  4                                :Value of N
 -6.99   7.92   2.34   0.32
 -0.44  -2.63  -1.18                :End of matrix T
```

9.3. Program Results

```
F08JEF Example Program Results

Eigenvalues
   -7.0037 -0.4059  2.0028  8.9968

Eigenvectors
          1       2       3       4
1    0.9995 -0.0109 -0.0167 -0.0255
2    0.0310  0.1627  0.3408  0.9254
3    0.0089  0.5170  0.7696 -0.3746
4    0.0014  0.8403 -0.5397  0.0509
```

Annexe B

Le protocole SHMEM

NAME

intro_shmem - Introduction to logically shared memory access routines

IMPLEMENTATION

UNICOS, UNICOS/mk, and IRIX systems

DESCRIPTION

The logically shared, distributed memory access (SHMEM) routines provide low-latency, high-bandwidth communication for use in highly parallelized scalable programs.

The SHMEM routines are data passing library routines similar to message passing library routines. They can be used as an alternative to message passing routines such as Message Passing Interface (MPI) or Parallel Virtual Machine (PVM). Like the message passing routines, the SHMEM routines pass data between cooperating parallel processes.

SHMEM routines can be used in programs that perform computations in separate address spaces and that explicitly pass data to and from different processes in the program. These processes are also called processing elements (PEs).

The SHMEM routines minimize the overhead associated with data passing requests, maximize bandwidth, and minimize data latency. Data latency is the period of time that starts when a PE initiates a transfer of data and ends when a PE can use the data.

SHMEM routines support remote data transfer through put operations, which transfer data to a different PE, and get operations, which transfer data from a different PE. Other operations supported are work-shared broadcast and reduction, barrier synchronization, and atomic memory operations. An atomic memory operation is an atomic read-and-update operation, such as a fetch-and-increment, on a remote or local data object. The value read is guaranteed to be the value of the data object just prior to the update.

SHMEM ROUTINES

The following SHMEM-related routines are available on UNICOS, UNICOS/mk, and IRIX systems. Using these routines enhances the portability of SHMEM programs across platforms.

* PE queries:

  C/C++ only: _num_pes(3I), _my_pe(3I)

  Fortran only: NUM_PES(3I), MY_PE(3I)

* Block data put routines:

  C/C++ and Fortran: shmem_put32, shmem_put64, shmem_put128

  C/C++ only: shmem_double_put, shmem_float_put, shmem_int_put, shmem_long_put, shmem_short_put

  Fortran only: shmem_complex_put, shmem_integer_put, shmem_logical_put, shmem_real_put

* Block data get routines:

  C/C++ and Fortran: shmem_get32, shmem_get64, shmem_get128

  C/C++ only: shmem_double_get, shmem_float_get, shmem_int_get, shmem_long_get, shmem_short_get

  Fortran only: shmem_complex_get, shmem_integer_get, shmem_logical_get, shmem_real_get

* Strided put routines:

  C/C++ and Fortran: shmem_iput32, shmem_iput64, shmem_iput128

  C/C++ only: shmem_double_iput, shmem_float_iput, shmem_int_iput, shmem_long_iput, shmem_short_iput

  Fortran only: shmem_complex_iput, shmem_integer_iput, shmem_logical_iput, shmem_real_iput

* Strided get routines:

  C/C++ and Fortran: shmem_iget32, shmem_iget64, shmem_iget128

  C/C++ only: shmem_double_iget, shmem_float_iget, shmem_int_iget, shmem_long_iget, shmem_short_iget

  Fortran only: shmem_complex_iget, shmem_integer_iget, shmem_logical_iget, shmem_real_iget

* Point-to-point synchronization routines:

  C/C++ only: shmem_int_wait, shmem_int_wait_until, shmem_long_wait, shmem_long_wait_until, shmem_longlong_wait, shmem_longlong_wait_until, shmem_short_wait, shmem_short_wait_until

  Fortran: shmem_int4_wait, shmem_int4_wait_until, shmem_int8_wait, shmem_int8_wait_until

* Barrier synchronization routines:

  C/C++ and Fortran: shmem_barrier_all, shmem_barrier

* Atomic memory fetch-and-operate (fetch-op) routines:

  C/C++ and Fortran: shmem_swap

* Reduction routines:

  C/C++ only: shmem_short_and_to_all, shmem_int_and_to_all, shmem_short_max_to_all, shmem_int_max_to_all, shmem_float_max_to_all, shmem_double_max_to_all, shmem_short_min_to_all, shmem_int_min_to_all, shmem_float_min_to_all, shmem_double_min_to_all, shmem_short_sum_to_all, shmem_int_sum_to_all, shmem_float_sum_to_all, shmem_double_sum_to_all, shmem_short_prod_to_all, shmem_int_prod_to_all, shmem_float_prod_to_all,

shmem_double_prod_to_all,
shmem_short_or_to_all, shmem_int_or_to_all,
shmem_short_xor_to_all, shmem_int_xor_to_all

Fortran only: shmem_int4_and_to_all, shmem_int8_and_to_all,
shmem_real4_max_to_all,
shmem_real8_max_to_all,
shmem_int4_max_to_all, shmem_int8_max_to_all,
shmem_real4_min_to_all,
shmem_real8_min_to_all,
shmem_int4_min_to_all, shmem_int8_min_to_all,
shmem_real4_sum_to_all,
shmem_real8_sum_to_all,
shmem_int4_sum_to_all, shmem_int8_sum_to_all,
shmem_real4_prod_to_all,
shmem_real8_prod_to_all,
shmem_int4_prod_to_all,
shmem_int8_prod_to_all, shmem_int4_or_to_all,
shmem_int8_or_to_all, shmem_int4_xor_to_all,
shmem_int8_xor_to_all

* Broadcast routines:

C/C++ and Fortran: shmem_broadcast32, shmem_broadcast64

* Generalized barrier synchronization routine:

C/C++ and Fortran: shmem_barrier

* Cache management routines:

C/C++ and Fortran: shmem_udcflush, shmem_udcflush_line

SHMEM ROUTINES ON UNICOS/mk AND IRIX SYSTEMS

The following routines are available only on UNICOS/mk and IRIX systems systems:

* Byte-granularity block put routines:

C/C++ and Fortran: shmem_putmem and shmem_getmem

Fortran only: shmem_character_put and shmem_character_get

* Collect routines:

C/C++ and Fortran: shmem_collect32, shmem_collect64,
shmem_fcollect32, shmem_fcollect64

* Atomic memory fetch-and-operate (fetch-op) routines:

C/C++ only: shmem_double_swap, shmem_float_swap,
shmem_int_cswap, shmem_int_fadd,
shmem_int_finc, shmem_int_swap,
shmem_long_cswap, shmem_long_fadd,
shmem_long_finc, shmem_long_swap,
shmem_longlong_cswap, shmem_longlong_fadd,
shmem_longlong_finc, shmem_longlong_swap

Fortran only: shmem_int4_cswap, shmem_int4_fadd,
shmem_int4_finc, shmem_int4_swap,
shmem_int8_swap, shmem_real4_swap,
shmem_real8_swap, shmem_int8_cswap

* Atomic memory operation routines:

Fortran only: shmem_int4_add, shmem_int4_inc

SHMEM ROUTINES ON UNICOS AND IRIX SYSTEMS

The following routines are available only on UNICOS and IRIX systems:

* Remote memory pointer function:

C/C++ and Fortran: shmem_ptr

* Reduction routines:

C/C++ only: shmem_longdouble_max_to_all,
shmem_longdouble_min_to_all,
shmem_longdouble_prod_to_all,
shmem_longdouble_sum_to_all

Fortran only: shmem_real16_max_to_all,
shmem_real16_min_to_all,
shmem_real16_prod_to_all,
shmem_real16_sum_to_all

SHMEM ROUTINES ON UNICOS AND UNICOS/mk SYSTEMS

The following routines are available only on UNICOS and UNICOS/mk systems:

* Generic put/iput and get/iget routines:

C/C++ and Fortran: shmem_put, shmem_get, shmem_iput, shmem_iget

* Reduction routines:

C/C++ only: shmem_complexf_sum_to_all,
shmem_complexd_sum_to_all,
shmem_complexf_prod_to_all,
shmem_complexd_prod_to_all

Fortran only: shmem_comp4_sum_to_all,
shmem_comp8_sum_to_all,
shmem_comp4_prod_to_all,
shmem_comp8_prod_to_all

* Generic broadcast routine:

C/C++ and Fortran: shmem_broadcast

SHMEM ROUTINES ON UNICOS/mk SYSTEMS

The following routines are available only on UNICOS/mk systems:

* Gather/scatter routines:

C/C++ and Fortran: shmem_ixput32, shmem_ixput, shmem_ixget32,
shmem_ixget

* Generic collect routines:

C/C++ and Fortran: shmem_collect, shmem_fcollect

* Atomic memory fetch-and-operate (fetch-op) routines:

C C++ only: shmem_short_finc, shmem_short_swap

* Atomic memory operation routines:

C C++ only: shmem_short_inc

REMOTELY ACCESSIBLE DATA OBJECTS

Typically, target or source arrays that reside on remote processing elements (PEs) are identified by passing the address of the corresponding data object on the local PE. The local existence of a corresponding data object implies that a data object is symmetric as described on this man page. Most uses of SHMEM routines employ symmetric data objects, but on UNICOS/mk systems, another class of data object, asymmetric accessible data objects, can also be passed to SHMEM routines.

Symmetric or asymmetric accessible data objects passed to SHMEM routines can be arrays or scalars. The symmetric and the asymmetric accessible addressing classes are as follows:

* A symmetric data object is one for which the local and remote addresses have a known relationship. You can use SHMEM routines to access remote symmetric data objects by using the address of the corresponding data object on the local PE.

  The following data objects are symmetric on UNICOS, UNICOS/mk, and IRIX systems:

  - Fortran data objects in common blocks or with the SAVE attribute. On IRIX systems, these data objects must not be defined in a dynamic shared object (DSO). On UNICOS systems the -a taskcommon compiler command line option must be specified.

  - Non-stack C and C++ variables. On IRIX systems, these data objects must not be defined in a DSO. On UNICOS systems the -h taskprivate compiler command line option must be specified.

  The following data objects are symmetric on UNICOS/mk and IRIX systems:

  - Fortran arrays allocated with shpalloc(3F)

  - C and C++ data allocated by shmalloc(3C)

  The following data objects are symmetric on UNICOS/mk systems:

  - C and C++ stack variables declared with a #pragma symmetric directive

  - Fortran stack variables declared with a !DIR$ SYMMETRIC directive

  The following data objects are symmetric on UNICOS systems:

  - Fortran arrays and scalars declared on the TASKCOMMON compiler directive

  - C/C++ scalar and array objects declared in #pragma taskcommon directives

  SHMEM collective routines that operate on the same data object on multiple PEs require that symmetric data objects be passed. This restriction is for algorithm simplicity and efficiency. These routines define the set of target PEs by the following triplet of arguments: PE_start, logPE_stride, and PE_size.

* On UNICOS/mk systems, an asymmetric accessible type of data object can be accessed from other PEs through SHMEM routines. An asymmetric accessible object is remotely accessible, but not symmetric. Therefore, these data objects can be accessed on UNICOS/mk systems though SHMEM routines only if their address is communicated between PEs first by use of a prelude SHMEM or other message passing library call.

  The following data objects are asymmetric accessible on UNICOS/mk systems:

  - C/C++ data allocated by malloc(3C) and C++ data allocated by the new operator

  - C/C++ variables with automatic or register storage class

  - Fortran arrays allocated with hpalloc(3F)

  - Fortran PE-private data objects on the stack

COLLECTIVE ROUTINES

Some SHMEM routines, for example, shmem_broadcast(3) and shmem_float_sum_to_all(3), are classified as collective routines because they distribute work across a set of PEs. They must be called concurrently by all PEs in the active set defined by the PE_start, logPE_stride, PE_size argument triplet. The following man pages describe the SHMEM collective routines:

* shmem_and(3)
* shmem_barrier(3)
* shmem_broadcast(3)
* shmem_collect(3)
* shmem_max(3)
* shmem_min(3)
* shmem_or(3)
* shmem_prod(3)
* shmem_sum(3)
* shmem_xor(3)

USING THE SYMMETRIC WORK ARRAY, pSync

Multiple pSync arrays are often needed if a particular PE calls a SHMEM collective routine twice without intervening barrier synchronization. Problems would occur if some PEs in the active set for call 2 arrive at call 2 before processing of call 1 is complete by all PEs in the call 1 active set. You can use shmem_barrier() or shmem_barrier_all(3) to perform a barrier synchronization between consecutive calls to SHMEM collective routines.

There are two special cases:

* The shmem_barrier(3) routine allows the same pSync array to be used on consecutive calls as long as the active PE set does not change.

* If the same collective routine is called multiple times with the same active set, the calls may alternate between two pSync arrays. The SHMEM routines guarantee that a first call is completely finished by all PEs by the time processing of a third call begins on any PE.

Because the SHMEM routines restore pSync to its original contents, multiple calls that use the same pSync array do not require that pSync be reinitialized after the first call.

SHMEM FUNCTION INLINING

Some SHMEM functions that can be called from C/C++ are defined in the form of macros in the mpp/shmem.h header file. These functions are inlined by default on some platforms. To deactivate the automatic inlining of SHMEM functions from C/C++, add the following option to your C C++ command line: -D_SHMEM_MACRO_OPT=0.

SHMEM APPLICATION PLACEMENT ON NUMA SYSTEMS

In non-uniform memory access (NUMA) systems, such as Origin series systems, SHMEM start-up processing ensures that the process associated with a SHMEM PE executes on a processor near the memory associated with a SHMEM PE.

The following environment variables allow you to control the placement of the SHMEM application on the system:

Variable	Description
SMA_DSM_OFF	When set to any value, deactivates processor-memory affinity control. When set, SHMEM processes run on any available processor, whether or not it is near the memory associated with that process.
SMA_DSM_VERBOSE	When set to any value, writes information about process and memory placement to stderr.
SMA_DSM_PPM	When set to an integer value, specifies the number of processors to be mapped to every memory. The default is 2.
SMA_DSM_TOPOLOGY	Specifies the shape of the set of hardware nodes on which the PE memories are allocated. Set this variable to one of the following values: Value — Action cube — A group of memory nodes that form a perfect hypercube. NPES/SMA_DSM_PPM must be a power of 2. free — Any group of memory nodes. The operating system attempts to place the group numbers close to one another. (Default)
PAGESIZE_DATA	Specifies the desired page size in kilobytes for program data areas. Specify an integer value. On Origin series systems, supported values include 16, 64, 256, and 1024.
SMA_INFO	Prints information about environment variables that can control libsma execution.
SMA_SYMMETRIC_SIZE	Specifies the size, in bytes, of symmetric memory. This is the size of static space plus per-PE symmetric heap size.
SMA_VERSION	Prints the libsma library release version.

USING dplace(1)

The environment variables described previously allow you to map SHMEM processes and memories with hardware processors and nodes. The dplace(1) command, which is available on Origin series systems, can give you additional control over application placement.

Perform the following steps to use the dplace(1) command with SHMEM programs:

* Create file placefile with these contents:

```
threads $NPES + 1
memories ($NPES +1)/2 in topology cube
distribute threads 1:$NPES across memories
```

* Execute your program with NPES set to the number of PEs. For example, to run with 4 PEs, invoke your program this way:

```
env NPES=4 dplace -place placefile a.out
```

INTEROPERABILITY

On UNICOS/mk and UNICOS systems, SHMEM routines can be used in conjunction with message passing routines in the same application.

COMPILING SHMEM CODES

The SHMEM routines reside in libsma.so on IRIX systems; they reside in libsma.a on UNICOS/mk and UNICOS systems.

The following sample command lines compile programs that include SHMEM routines:

* UNICOS/mk systems:

```
cc c_program.c
CC cplusplus_program.c
f90 fortran_program.f
```

* IRIX systems:

```
cc -64 c_program.c -lsma
CC -64 cplusplus_program.c -lsma
f90 -64 -LANG:recursive=on fortran_program.f -lsma
f77 -64 -LANG:recursive=on fortran_program.f -lsma
```

* UNICOS systems:

```
module load mpt; cc -htaskprivate c_program.c
module load mpt; f90 -ataskcommon fortran_program.f
```

NOTES

On UNICOS/mk systems, the SHMEM software is packaged with CrayLibs. On UNICOS and IRIX systems, the SHMEM software is packaged with the Message Passing Toolkit.

Some of the compiler directives and command line options mentioned on this man page may not be available to you. Please consult your compiler's documentation for information on directives and options.

ENVIRONMENT VARIABLES

For information on environment variables that affect SHMEM routines,

See the SHMEM APPLICATION PLACEMENT ON NUMA SYSTEMS section of this man page

EXAMPLES

Example 1. The following Fortran SHMEM program runs on UNICOS, UNICOS/mk, and IRIX systems:

```
      PROGRAM REDUCTION
      REAL VALUES, SUM
      COMMON /C/ VALUES
      REAL WORK
      CALL START_PES(0)
      VALUES = MY_PE()
      CALL SHMEM_BARRIER_ALL        ! Synchronize all PEs
      SUM = 0.0
      DO I = 0,NUM_PES()-1
         CALL SHMEM_REAL_GET(WORK, VALUES, 1, I)    ! Get next value
         SUM = SUM + WORK                           ! Sum it
      ENDDO
      PRINT*,'PE ',MY_PE(),' COMPUTED       SUM=',SUM
      CALL SHMEM_BARRIER_ALL
      END
```

Since start_pes(3) is called with a value of 0, the number of PEs used to run the program is specified by the NPES environment variable on UNICOS and IRIX systems. The number of PEs is specified by the -n option on the mpprun(1) command on UNICOS/mk systems.

This Fortran program directs all PEs to sum simultaneously the numbers in the VALUES variable across all PEs. By executing the program using the following command line, you can you can run the program with 4 PEs:

```
env NPES=4 a.out
```

On UNICOS/mk systems, enter the following command to execute the program:

```
mpprun -n 4 a.out
```

Example 2. The following C SHMEM program runs on UNICOS, UNICOS/mk, and IRIX systems:

```
#include <mpp/shmem.h>
main()
{
   long source[10] = { 1, 2, 3, 4, 5,
                       6, 7, 8, 9, 10 };
static long target[10];
start_pes(0);
if (_my_pe() == 0) {
   /* put 10 words into target on PE 1 */
   shmem_long_put(target, source, 10, 1);
}
shmem_barrier_all();  /* sync sender and receiver */
if (_my_pe() == 1)
   shmem_udcflush();          /* needed on T90  */
   printf("target[0] on PE %d is %d\n", _my_pe(), target[0]);
}
```

In this C program, PE 0 sends 10 integers to the target array on PE 1. By executing the program using the following command line, you can you can run the program with 2 PEs:

```
env NPES=2 a.out
```

On UNICOS/mk systems, enter the following command to execute the program:

```
mpprun -n 2 a.out
```

SEE ALSO

The following command is available only on IRIX systems:

dplace(1)

The following man pages and Cray Research reference manuals also contain information on SHMEM routines. Not all routines are available on all systems. See the specific man pages for implementation information.

CC(1), cld(1), f90(1), f90(1M), mpprun(1)

fast_shmem_inc(3), shmem_add(3), shmem_and(3), shmem_barrier(3), shmem_barrier_all(3), shmem_broadcast(3), shmem_cache(3), shmem_collect(3), shmem_cswap(3), shmem_event(3), shmem_fadd(3), shmem_fence(3), shmem_finc(3), shmem_get(3), shmem_iget(3), shmem_inc(3), shmem_iput(3), shmem_ixput(3), shmem_lock(3), shmem_max(3), shmem_min(3), shmem_mswap(3), shmem_my_pe(3), shmem_or(3), shmem_prod(3), shmem_put(3), shmem_quiet(3), shmem_short_g(3) shmem_short_p(3), shmem_stack(3), shmem_sum(3), shmem_swap(3), shmem_wait(3), shmem_xor(3), start_pes(3)

shmalloc(3C)

hpalloc(3F), shmalloc(3F), shpalloc(3F)

MY_PE(3I), NUM_PES(3I)

For information on using SHMEM routines to optimize Fortran program performance on UNICOS/mk systems, see CRAY T3E Fortran Optimization Guide, publication SG-2518.

For information on using SHMEM routines with message passing routines, see the Message Passing Toolkit: PVM Programmer's Manual, publication SR-2196, or the Message Passing Toolkit: MPI Programmer's Manual, publication SR-2197.

Application Programmer's Library Reference Manual, publication SR-2165, for the printed version of this man page.

Annexe C

Programme parallèle 1

```
!*****************************************************************************
PROGRAM main
!*****************************************************************************

  DESCRIPTIF : diagonalise des chaines de spins

  DONNEES : lit dans le fichier 'param.txt' la dimension
            totale de l'espace, le nb d'elements non nuls de l'hamiltonien,
            le spin des atomes, et 'NbIterations' de la methode de lanczos

!-----------------------------------------------------------------------------

  USE files
  USE diagonalisation

  IMPLICIT NONE

  INTEGER    NbSpins, N, NbIterations
  REAL*8     Delta
  REAL*8     S

  INTEGER*8 pSync(SHMEM_BCAST_SYNC_SIZE)

  !DIR$ SYMMETRIC pSync
  INTRINSIC MY_PE

!-----------------------------------------------------------------------------

  !! lecture du fichier de donnees
  IF (MY_PE() == 0) THEN
     CALL readData(S, NbSpins, Delta, NbIterations)
  END IF

  !  synchronisation des processeurs
  DO i = 1, SHMEM_BCAST_SYNC_SIZE
     pSync(i) = SHMEM_SYNC_VALUE
  END DO

  !! broadcaste les parametres du systeme
  CALL SHMEM_BARRIER_ALL
  CALL SHMEM_BROADCAST(S, S, 1, 0, 0, 0, n$pes, pSync)
  CALL SHMEM_BARRIER_ALL
  CALL SHMEM_BROADCAST(NbSpins, NbSpins, 1, 0, 0, 0, n$pes, pSync)
  CALL SHMEM_BARRIER_ALL
  CALL SHMEM_BROADCAST(Delta, Delta, 1, 0, 0, 0, n$pes, pSync)
  CALL SHMEM_BARRIER_ALL
  CALL SHMEM_BROADCAST(NbIterations, NbIterations, 1, 0, 0, 0, n$pes, pSync)

  !! calcul du nombre d'etats
  N = 2**NbSpins

  !! on traite le cas spin 1/2
  IF (S == 0.5) THEN
     CALL diagoDemi(NbSpins, N, Delta, NbIterations)
  END IF

  !  ecriture du fichier de commandes gnuplot
  CALL writePlot(NbIterations)

!*****************************************************************************
END PROGRAM main
!*****************************************************************************
```

```
!*****************************************************************************
MODULE files
!*****************************************************************************

! DESCRIPTIF : ce module contient des outils pour les entrees-sorties

!-----------------------------------------------------------------------------

CONTAINS

!=============================================================================
SUBROUTINE readData(S, NbSpins, Delta, NbIterations)
!=============================================================================

! DESCRIPTIF : lit le fichier de config

  !---------------------------------------------------------------------------

  IMPLICIT NONE

  INTEGER, INTENT(out) :: NbSpins, NbIterations
  REAL,    INTENT(out) :: S
  REAL*8,  INTENT(out) :: Delta

!-----------------------------------------------------------------------------

  !! lecture des parametres
  READ *, S
  READ *, NbSpins
  READ *, Delta
  READ *, NbIterations

!=============================================================================
END SUBROUTINE readData
!=============================================================================



!=============================================================================
SUBROUTINE writePlot(NbIterations)
!=============================================================================

! DESCRIPTIF : ecrit le fichier de commandes gnuplot

!-----------------------------------------------------------------------------

  IMPLICIT NONE

  CHARACTER (*), PARAMETER :: namPlot = 'bornes.gnuplot'
  INTEGER,       PARAMETER :: numPlot = 1

  INTEGER, INTENT(in) :: NbIterations

!-----------------------------------------------------------------------------

  !! ouverture du fichier de commandes
  OPEN (unit=numPlot, file=namPlot, form='formatted', status='replace')

  !! on ecrit les commandes gnuplot
  WRITE (numPlot, *) 'set data style lines'
  WRITE (numPlot, *) 'set xlabel "Iterations"'
  WRITE (numPlot, *) 'set ylabel "Energie"'
  WRITE (numPlot, *) 'plot [x=1:',NbIterations,'] [-5:3] "bornes.txt" using 1 notitle'
  WRITE (numPlot, *) 'replot "bornes.txt" using 2 notitle'
  WRITE (numPlot, *) 'replot "bornes.txt" using 3 notitle'
  WRITE (numPlot, *) 'replot "bornes.txt" using 4 notitle'
  WRITE (numPlot, *) 'replot "bornes.txt" using 5 notitle'
  WRITE (numPlot, *) 'replot "bornes.txt" using 6 notitle'

  !! fermeture du fichier de commandes
  CLOSE (numPlot)

!=============================================================================
END SUBROUTINE writePlot
!=============================================================================



!=============================================================================
SUBROUTINE spectre(N, eigenvalues, iteration)
!=============================================================================

!! DESCRIPTIF : selectionne les "bonnes" valeurs propres

!-----------------------------------------------------------------------------

  IMPLICIT NONE

  CHARACTER (*), PARAMETER :: namSpec = 'spectre.txt'
  INTEGER,       PARAMETER :: numSpec = 2

  INTEGER, INTENT(in) :: N, iteration
  REAL*8,  INTENT(in) :: eigenvalues(N)

!-----------------------------------------------------------------------------

  !! ouverture du fichier de commandes
  OPEN (unit=numSpec, file=namSpec, form='formatted', status='replace')

  !! enregistrement du spectre non degenere
  WRITE (numSpec, *) 'SPECTRE ITERATION #', iteration
  WRITE (numSpec, '(f14.5)')  eigenvalues(1:iteration)

  !! fermeture du fichier de donnees
  CLOSE (numSpec)

!=============================================================================
END SUBROUTINE spectre
!=============================================================================



!=============================================================================
SUBROUTINE extreme(numBound, NbIterations, eigenvalues, iteration)
!=============================================================================

!! DESCRIPTIF : selectionne les "bonnes" valeurs propres

!-----------------------------------------------------------------------------

  IMPLICIT NONE

  INTEGER, INTENT(in) :: NbIterations, iteration, numBound
  REAL*8,  INTENT(in) :: eigenvalues(NbIterations)

!-----------------------------------------------------------------------------

  IF (iteration >= 4) THEN
```

```
    WRITE (numBound, *) eigenvalues(1), ' ',&
                       &eigenvalues(2), ' ',&
                       &eigenvalues(3), ' ',&
                       &eigenvalues(iteration-2), ' ',&
                       &eigenvalues(iteration-1), ' ',&
                       &eigenvalues(iteration)
  END IF

!=============================================================================
END SUBROUTINE extreme
!=============================================================================


!*****************************************************************************
END MODULE files
!*****************************************************************************
```

```
!*****************************************************************************
MODULE diagonalisation
!*****************************************************************************

  ! DESCRIPTIF : tridiagonalisation a la Lanczos et recherche des valeurs
  !              propres a la NAG ; les ecrit dans le fichier 'namResult'
  !
  ! DONNEES : 'NbIterations' est le nb max d'iterations pour lanczos
  !           'numResult' et 'namResult' caracterisent le fichier de resultats

!-----------------------------------------------------------------------------

#include "mpp_shmem.fh"

CONTAINS

!=============================================================================
SUBROUTINE diagoDemi(NbSpins, N, Delta, NbIterations)
!=============================================================================

  ! DESCRIPTIF : diagonalisation dans le cas d'un spin 1/2

!-----------------------------------------------------------------------------

  USE produit
  USE files

  IMPLICIT NONE

  CHARACTER (*), PARAMETER :: namResult = 'result.txt', namBound = 'bornes.txt'
  INTEGER,       PARAMETER :: numResult = 99, numBound = 98

  INTEGER, INTENT(in) :: NbSpins, N, NbIterations
  REAL*8,  INTENT(in) :: Delta

  REAL*8  alpha, beta, fondamental, excite
  REAL*8  diagonale(NbIterations), sousDiagonale(NbIterations - 1)
  REAL*8  work(2 * NbIterations), A(NbIterations), B(NbIterations)
  REAL*8  eigenvalues(NbIterations), eigenvectors(NbIterations, NbIterations)
  REAL*8  qOld(N), qNew(N), init(N), r(N)
  INTEGER iteration, info, partiel, me

!DIR$ SYMMETRIC r
  INTRINSIC MY_PE

!-----------------------------------------------------------------------------

  ! initialisation du vecteur de lanczos
  CALL RANDOM_NUMBER(init)
  init = init - 0.5
  qNew = init / SQRT(DOT_PRODUCT(init, init))
  me = MY_PE()

  ! ouverture des fichiers de donnees
  IF (me == 0) THEN
     OPEN (unit=numResult, file=namResult, form='formatted', status='replace')
  END IF

  ! boucle lanczos : voir algorithme
  DO iteration = 1, NbIterations

     ! multiplication creuse : r = H.q
     CALL multDemi(NbSpins, N, Delta, qNew, r)

     ! tridiagonalisation lanczos
     IF (iteration > 1) THEN
        r = r - beta * qOld
     END IF
     alpha = DOT_PRODUCT(r, qNew)
     r = r - alpha * qNew
     beta = SQRT(DOT_PRODUCT(r, r))
     IF (iteration < NbIterations) THEN
        qOld = qNew
        qNew = r / beta
        sousDiagonale(iteration) = beta
     END IF
     diagonale(iteration) = alpha

     ! procedure de diagonalisation NAG
     A(1:iteration) = diagonale(1:iteration)
     B(1:iteration - 1) = sousDiagonale(1:iteration - 1)
     CALL F08JEE('I', iteration, A, B, eigenvectors, NbIterations, work, info)
     IF (info /= 0) STOP 'plantage NAG...'
     eigenvalues = A

     ! enregistrement des energies extremes et enregistrement du resultat
     IF (me == 0) THEN
        PRINT *, 'ITERATION #', iteration, ' sur ', NbIterations
        WRITE (numResult, *) 'SPECTRE ITERATION #', iteration
        WRITE (numResult, '(f14.5)') eigenvalues(1:iteration)
     END IF

     ! terminaison
     IF (iteration == N) THEN
        EXIT
     END IF

  END DO

  ! enregistrement du spectre et fermeture des fichiers de donnees
  IF (me == 0) THEN
     CLOSE (numResult)
     CALL spectre(N, eigenvalues, iteration)
  END IF

!=============================================================================
END SUBROUTINE diagoDemi
!=============================================================================


!*****************************************************************************
END MODULE diagonalisation
!*****************************************************************************
```

```
!*****************************************************************************
MODULE produit
!*****************************************************************************

! DESCRIPTIF : ce module decrit le calcul de produits matrice-vecteur

!-----------------------------------------------------------------------------

INCLUDE 'mpp_shmem.fh'

CONTAINS

!=============================================================================
SUBROUTINE multDemi(NbSpins, N,  Delta, u, v)
!=============================================================================

  DESCRIPTIF : on construit l'action de l'hamiltonien de Heisenberg pour S=1/2

!-----------------------------------------------------------------------------

  IMPLICIT NONE

  REAL*8, PARAMETER :: DEMI = 0.5_8, QUART = 0.25_8

  INTEGER, INTENT(in)  :: N, NbSpins
  REAL*8,  INTENT(in)  :: u(N), Delta
  REAL*8,  INTENT(out) :: v(N)

  INTEGER   i, j, me, etat, masque, etatPartiel
  INTEGER*8 pSync(SHMEM_REDUCE_SYNC_SIZE)
  REAL*8    pWrk(N/2+1)

  !DIR$ SYMMETRIC pWrk, pSync
  INTRINSIC MY_PE

!-----------------------------------------------------------------------------

  !! initialisations algo
  v = 0.0

  !! initialisations //
  me = MY_PE()
  IF (nSpes /= NbSpins) STOP 'Pas le bon nb de processeurs...'

  !! calcul d'un terme de l'hamiltonien
  IF (me == nSpes - 1) THEN

     !! action de Sn.S1
     DO etatPartiel = 0, N/4 - 1
        etat = ISHFT(etatPartiel, 1)

        !! etat --
        i = IBCLR(IBCLR(etat, me), 0) + 1
        v(i) = v(i) + QUART * Delta * u(i)

        !! etat -+
        i = IBCLR(IBSET(etat, me), 0) + 1
        j = IBSET(IBCLR(etat, me), 0) + 1
        v(i) = v(i) - QUART * u(i) + DEMI * u(j)

        !! etat +-
        i = IBSET(IBCLR(etat, me), 0) + 1
        j = IBCLR(IBSET(etat, me), 0) + 1
        v(i) = v(i) - QUART * u(i) + DEMI * u(j)

        !! etat ++
        i = IBSET(IBSET(etat, me), 0) + 1
        v(i) = v(i) + QUART * Delta * u(i)

     END DO

  ELSE

     !! action de Si.Si+1
     DO etatPartiel = 0, N/4 - 1
        masque = IBSET(0, me) - 1
        etat = ISHFT(IAND(NOT(masque), etatPartiel), 2) + IAND(masque, etatPartiel)

        !! etat --
        i = IBCLR(IBCLR(etat, me), me + 1) + 1
        v(i) = v(i) + QUART * Delta * u(i)

        !! etat -+
        i = IBCLR(IBSET(etat, me), me + 1) + 1
        j = IBSET(IBCLR(etat, me), me + 1) + 1
        v(i) = v(i) - QUART * u(i) + DEMI * u(j)

        !! etat +-
        i = IBSET(IBCLR(etat, me), me + 1) + 1
        j = IBCLR(IBSET(etat, me), me + 1) + 1
        v(i) = v(i) - QUART * u(i) + DEMI * u(j)

        !! etat ++
        i = IBSET(IBSET(etat, me), me + 1) + 1
        v(i) = v(i) + QUART * Delta * u(i)

     END DO

  END IF

  !! synchronisation des processeurs
  DO i = 1, SHMEM_REDUCE_SYNC_SIZE
     pSync(i) = SHMEM_SYNC_VALUE
  END DO
  CALL SHMEM_BARRIER_ALL

  !! reduction de l'hamiltonien sur les processeurs
  CALL SHMEM_REAL8_SUM_TO_ALL(v, v, N, 0, 0, nSpes, pWrk, pSync)

!=============================================================================
END SUBROUTINE multDemi
!=============================================================================



!*****************************************************************************
END MODULE produit
!*****************************************************************************
```

Annexe D

Programme parallèle 2

```
!*****************************************************************************
MODULE diagonalisation
!*****************************************************************************

   DESCRIPTIF : tridiagonalisation a la Lanczos et recherche des valeurs
                propres a la NAG ; les ecrit dans le fichier 'namResult'

   DONNEES : 'NbIterations' est le nb max d'iterations pour lanczos
             'numResult' et 'namResult' caracterisent le fichier de resultats

!-----------------------------------------------------------------------------

INCLUDE "mpp_shmem.fh"

CONTAINS

!=============================================================================
SUBROUTINE diagoDemi(NbSpins, N, Delta, NbIterations)
!=============================================================================

   DESCRIPTIF : diagonalisation dans le cas d'un spin 1/2

!-----------------------------------------------------------------------------

  USE produit
  USE files

  IMPLICIT NONE

  CHARACTER (*), PARAMETER :: namResult = 'result.txt', namBound = 'bornes.txt'
  INTEGER,       PARAMETER :: numResult = 99, numBound = 98

  INTEGER, INTENT(in) :: NbSpins, N, NbIterations
  REAL*8,  INTENT(in) :: Delta

  REAL*8  alpha, beta, fondamental, excite
  REAL*8  diagonale(NbIterations), sousDiagonale(NbIterations - 1)
  REAL*8  work(2 * NbIterations), A(NbIterations), B(NbIterations)
  REAL*8  eigenvalues(NbIterations), eigenvectors(NbIterations, NbIterations)
  REAL*8  qOld(N), qNew(N), init(N), r(N)
  INTEGER iteration, info, partiel, me

  !DIR$ SYMMETRIC qNew
  INTRINSIC MY_PE

!-----------------------------------------------------------------------------

  !! initialisation du vecteur de lanczos
  CALL RANDOM_NUMBER(init)
  init = init - 0.5
  qNew = init / SQRT(parDot(init, init, N))
  me = MY_PE()

  !! ouverture des fichiers de donnees
  IF (me == 0) THEN
     OPEN (unit=numResult, file=namResult, form='formatted', status='replace')
  END IF

  !! boucle lanczos : voir algorithme
  DO iteration = 1, NbIterations

     !! multiplication creuse : r = H.q
     CALL multDemi(NbSpins, N, qNew, r)

     !! tridiagonalisation lanczos
     IF (iteration > 1) THEN
        r = r - beta * qOld
     END IF
     alpha = parDot(r, qNew, N)
     r = r - alpha * qNew
     beta = SQRT(parDot(r, r, N))
     IF (iteration < NbIterations) THEN
        qOld = qNew
        qNew = r / beta
        sousDiagonale(iteration) = beta
     END IF
     diagonale(iteration) = alpha

     !! procedure de diagonalisation NAG
     A(1:iteration) = diagonale(1:iteration)
     B(1:iteration - 1) = sousDiagonale(1:iteration - 1)
     CALL F08JEE('I', iteration, A, B, eigenvectors, NbIterations, work, info)
     IF (info /= 0) STOP 'plantage NAG...'
     eigenvalues = A

     !! enregistrement des energies extremes et enregistrement du resultat
     IF (me == 0) THEN
        PRINT *, 'ITERATION #', iteration, ' sur ', NbIterations
        WRITE (numResult, *) 'SPECTRE ITERATION #', iteration
        WRITE (numResult, '(f14.5)') eigenvalues(1:iteration)
     END IF

     !! terminaison
     IF (iteration == (n$pes * N)) THEN
        EXIT
     END IF

  END DO

  !! enregistrement du spectre et fermeture des fichiers de donnees
  IF (me == 0) THEN
     CLOSE (numResult)
     CALL spectre(N, eigenvalues, iteration)
  END IF

!=============================================================================
END SUBROUTINE diagoDemi
!=============================================================================



!*****************************************************************************
END MODULE diagonalisation
!*****************************************************************************
```

```
!*****************************************************************************
MODULE produit
!*****************************************************************************

! DESCRIPTIF : ce module contient des outils pour le calcul matriciel

!-----------------------------------------------------------------------------

INCLUDE "mpp/shmem.fh"

CONTAINS

!=============================================================================
FUNCTION parDot(u, v, N)
!=============================================================================

! DESCRIPTIF : effectue le produit scalaire u.v en parallele

!-----------------------------------------------------------------------------

  IMPLICIT NONE

  INTEGER, INTENT(in) :: N
  REAL*8,  INTENT(in) :: u(N)
  REAL*8,  INTENT(in) :: v(N)

  INTEGER*8 pSync(SHMEM_REDUCE_SYNC_SIZE)
  REAL*8    pWrk(SHMEM_REDUCE_MIN_WRKDATA_SIZE)
  REAL*8    dot, parDot
  INTEGER   i

  !DIR$ SYMMETRIC dot, pWrk, pSync
  INTRINSIC MY_PE

!-----------------------------------------------------------------------------

  !! calcul local du produit scalaire
  dot = DOT_PRODUCT(u, v)

  !! synchronisation des processeurs
  DO i = 1, SHMEM_REDUCE_SYNC_SIZE
     pSync(i) = SHMEM_SYNC_VALUE
  END DO
  CALL SHMEM_BARRIER_ALL

  !! reduction du produit scalaire sur les processeurs
  CALL SHMEM_REAL8_SUM_TO_ALL(parDot, dot, 1, 0, 0, n$pes, pWrk, pSync)

!=============================================================================
END FUNCTION parDot
!=============================================================================



!=============================================================================
SUBROUTINE multDemi(NbSpins, N,  u, v)
!=============================================================================

! DESCRIPTIF : on construit l'action de l'hamiltonien de Heisenberg pour S=1/2

!-----------------------------------------------------------------------------

  IMPLICIT NONE

  REAL*8, PARAMETER :: DEMI = 0.5_8, QUART = 0.25_8

  INTEGER, INTENT(in)  :: N, NbSpins
  REAL*8,  INTENT(in)  :: u(N)
  REAL*8,  INTENT(out) :: v(N)

  INTEGER i, j, spin, etat, me, masque, etatPartiel
  REAL*8  uj

  INTRINSIC MY_PE

!-----------------------------------------------------------------------------

  !! initialisations
  me = MY_PE()
  v(:) = 0.0

  !! action des Si.Si+1 - DD
  DO etatPartiel = 0, N/4 - 1
     DO spin = 0, NbSpins-4

        masque = IBSET(0, spin) - 1
        etat = ISHFT(IAND(NOT(masque), etatPartiel), 2) + IAND(masque, etatPartiel)

        !! etat --
        i = IBCLR(IBCLR(etat, spin), spin + 1) + 1
        v(i) = v(i) + QUART * u(i)

        !! etat -+
        i = IBCLR(IBSET(etat, spin), spin + 1) + 1
        j = IBSET(IBCLR(etat, spin), spin + 1) + 1
        v(i) = v(i) - QUART * u(i) + DEMI * u(j)

        !! etat +-
        i = IBSET(IBCLR(etat, spin), spin + 1) + 1
        j = IBCLR(IBSET(etat, spin), spin + 1) + 1
        v(i) = v(i) - QUART * u(i) + DEMI * u(j)

        !! etat ++
        i = IBSET(IBSET(etat, spin), spin + 1) + 1
        v(i) = v(i) + QUART * u(i)

     END DO
  END DO

  !! action de Sn.S1 - GD
  DO etatPartiel = 0, N/2 - 1

     etat = ISHFT(etatPartiel, 1)

     !! etat --
     IF (me == 0 .OR. me == 1) THEN
        i = IBCLR(etat, 0) + 1
        v(i) = v(i) + QUART * u(i)
     END IF

     !! etat -+
     IF (me == 0) THEN
        i = IBSET(etat, 0) + 1
        j = IBCLR(etat, 0) + 1
        CALL SHMEM_GET(uj, u(j), 1, 2)
        v(i) = v(i) - QUART * u(i) + DEMI * uj
     END IF
     IF (me == 1) THEN
```

```
      i = IBSET(etat, 0) + 1
      j = IBCLR(etat, 0) + 1
      CALL SHMEM_GET(uj, u(j), 1, 3)
      v(i) = v(i) - QUART * u(i) + DEMI * uj
   END IF

      etat +-
   IF (me == 2) THEN
      i = IBCLR(etat, 0) + 1
      j = IBSET(etat, 0) + 1
      CALL SHMEM_GET(uj, u(j), 1, 0)
      v(i) = v(i) - QUART * u(i) + DEMI * uj
   END IF
   IF (me == 3) THEN
      i = IBCLR(etat, 0) + 1
      j = IBSET(etat, 0) + 1
      CALL SHMEM_GET(uj, u(j), 1, 1)
      v(i) = v(i) - QUART * u(i) + DEMI * uj
   END IF

   !! etat ++
   IF (me == 2 .OR. me == 3) THEN
      i = IBSET(etat, 0) + 1
      v(i) = v(i) + QUART * u(i)
   END IF

END DO

!! action de Sn-1.Sn-2 - GG
DO etat = 0, N-2 - 1

   !! etat --
   IF (me == 0 .OR. me == 2) THEN
      i = IBCLR(etat, NbSpins-3) + 1
      v(i) = v(i) + QUART * u(i)
   END IF

   !! etat -+
   IF (me == 0) THEN
      i = IBSET(etat, NbSpins-3) + 1
      j = IBCLR(etat, NbSpins-3) + 1
      CALL SHMEM_GET(uj, u(j), 1, 1)
      v(i) = v(i) - QUART * u(i) + DEMI * uj
   END IF
   IF (me == 2) THEN
      i = IBSET(etat, NbSpins-3) + 1
      j = IBCLR(etat, NbSpins-3) + 1
      CALL SHMEM_GET(uj, u(j), 1, 3)
      v(i) = v(i) - QUART * u(i) + DEMI * uj
   END IF

   !! etat +-
   IF (me == 1) THEN
      i = IBCLR(etat, NbSpins-3) + 1
      j = IBSET(etat, NbSpins-3) + 1
      CALL SHMEM_GET(uj, u(j), 1, 0)
      v(i) = v(i) - QUART * u(i) + DEMI * uj
   END IF
   IF (me == 3) THEN
      i = IBCLR(etat, NbSpins-3) + 1
      j = IBSET(etat, NbSpins-3) + 1
      CALL SHMEM_GET(uj, u(j), 1, 2)
      v(i) = v(i) - QUART * u(i) + DEMI * uj
   END IF

   !! etat ++
   IF (me == 1 .OR. me == 3) THEN
      i = IBSET(etat, NbSpins-3) + 1
      v(i) = v(i) + QUART * u(i)
   END IF

END DO

!! action de Sn.Sn-1 - GG
DO etat = 0, N - 1

   !! etat --
   IF (me == 0) THEN
      i = etat + 1
      v(i) = v(i) + QUART * u(i)
   END IF

   !! etat -+
   i = etat + 1
   IF (me == 1) THEN
      j = i
      CALL SHMEM_GET(uj, u(j), 1, 2)
      v(i) = v(i) - QUART * u(i) + DEMI * uj
   END IF

   !! etat +-
   i = etat + 1
   IF (me == 2) THEN
      j = i
      CALL SHMEM_GET(uj, u(j), 1, 1)
      v(i) = v(i) - QUART * u(i) + DEMI * uj
   END IF

   !! etat ++
   IF (me == 3) THEN
      i = etat + 1
      v(i) = v(i) + QUART * u(i)
   END IF

END DO

!=============================================================================
END SUBROUTINE multDemi
!=============================================================================



!*****************************************************************************
END MODULE produit
!*****************************************************************************
```

CEA

DEMANDE D'AUTORISATION EN VUE D'UNE PUBLICATION OU D'UNE COMMUNICATION

Direction: DSM
Centre: Saclay
Service: SPhT
Réf: T98/075

Titre original du document: Calcul de chaînes de spins quantiques sur ordinateur parallèlle

AUTEURS	AFFILIATION	Dept/Serv/Sect	VISA
J. Lamarcq		SPhT	

Nature du document

Périodique	Conf./Congrès	Ouvrage	Rapport	Thèse	Cours	Mémoire de stage	Autre
☐	☐	☐	☒	☐	☐	☐	☐

Rapport de stage de fin d'études à Supélec, avril-juin 1998

Langue: Français

Les visas portés ci-dessous attestent que la qualité scientifique et technique de la publication proposée a été vérifiée et que la présente publication ne divulgue pas d'information brevetable, commercialement utilisable ou classée.

	Sigle	Nom	Date	Visa	Observation
Chef de Service					
Chef de Département	SPhT	J. Zinn-Justin	10/07/98	J. Zinn-Justin	