MASTER EN SCIENCES INFORMATIQUES

Les Tests ET1. Le standard. Des interprétations. Deux implémentations. Exploitation des résultats.

Colin, Jean-Noël

Award date:
1989

Awarding institution:
Universite de Namur

Année académique 1988-1989.

Les Tests ET1.

Le standard.
Des interprétations.
Deux implémentations.
Exploitation des résultats.

Mémoire présenté en vue
de l'obtention du diplôme
de licencié et maître en
informatique par
Jean-Noël Colin.

Promoteur du mémoire: Mme le professeur M. Noirhomme.

Facultés Universitaires Notre-Dame de la Paix

Namur

Résumé.

Le but de ce travail est de présenter le concept des tests ET1, de la manière la plus claire et la plus précise possible. En même temps, nous essaierons de trouver des alternatives au standard proposé, tout en respectant l'esprit dans lequel le benchmark ET1 a été développé. En plus de cette présentation, nous montrerons la manière dont nous avons implémenté les tests ET1 dans deux environnements différents. Nous nous attacherons ensuite à isoler les facteurs importants qui permettent d'expliquer les différences de performances enregistrées lors de l'exécution des tests sur les deux ordinateurs. Une conclusion terminera ce travail.

Abstract.

The object of this paper is to present as clearly and accurately as possible the concept of the ET1 or TP1 benchmark. At the same time, we 'll try to show the lacks of the original benchmark, and to find alternatives to the proposed implementation guidelines, while respecting the mind of the creators of the ET1. After this presentation, we will show the way in wich we have developped the ET1 tests, in two different environments. We will then look for the main factors that can explain the differences between the performances measured on both systems, during the execution of the ET1 tests. A conclusion will end this work.

Table des matières.

1. Introduction.

Depuis plusieurs années, on peut constater l'avènement des machines OLTP (On-Line Transaction Processing). Et devant cette poussée, on est forcé d'admettre que les benchmarks utilisés ne fournissaient qu'une vue restreinte des performances des machines [1].

Face à ces limitations, en 1985, un groupe de travail, au sein de la firme Tandem (dont l'architecture des machines est entièrement conçue pour le traitement en temps réel) a décidé d'élaborer une série de benchmarks qui permettraient de mettre en lumière des aspects oubliés par les benchmarks existants. L'idée sous-jacente à cette recherche était d'exprimer des résultats de tests de performance, non plus en Mips, en Mbytes ou en Gbytes par seconde, mais en termes plus compréhensibles pour l'utilisateur qui derrière son terminal utilise une application OLTP, et voudrait que ses requêtes soient rapidement satisfaites, que ses fichiers soient rapidement triés, et que ses applications de mise-à-jour ne prennent pas trop de temps. Le travail de ce groupe a abouti au benchmark ET1 ou TP1 (pour Transaction Processing), défini dans l'article paru dans Datamation: "A Measure of Transaction Processing Power" [1], que nous avons considéré comme définition standard des tests ET1 pour ce mémoire.

En première analyse, il semble que ces tests aient été développés dans un environnement Tandem, ce qui pose de nombreuses difficultés lorsque l'on veut les implémenter sur différentes machines. De plus, il n'y a guère de documents susceptibles de venir en aide à des personnes désirant implémenter un ET1.

Un premier objectif de ce travail est d'approfondir le concept ET1. Nous en présenterons la définition standard; ensuite, nous essaierons de mettre en lumière certains des éléments de la définition des tests qui peuvent être sujets à interprétation. Nous essaierons tout au cours de ce travail d'isoler ces éléments, et d'y donner diverses solutions alternatives.

En même temps, nous essaierons de dégager une démarche générale, qui permette d'implémenter un ET1 sur des machines différentes, dans des environnements différents, et d'exploiter les différents résultats fournis, afin de pouvoir en tirer des conclusions quant aux performances des machines testées. Cette démarche sera guidée par l'article [9].

Enfin, dans un but plus pratique, nous exposerons la manière dont nous avons mené des tests ET1, sur deux machines différentes; nous essaierons de mettre en évidence les facteurs de différences entre ces machines qui permettraient d'expliquer les écarts entre les résultats obtenus.

Selon certains bruits, le travail entrepris en 1985 par Tandem, serait à ce jour poursuivi et aurait abouti à la définition d'un second jeu de benchmarks, appelés ET2. Nous ne disposons malheureusement pas de plus d'information à ce sujet.

2. Présentation du cadre de travail.

Le développement de ce mémoire a été réalisé en deux temps. De septembre 1988 à janvier 1989, nous avons effectué un stage auprès de la société Trasys, membre du groupe Tractebel, dont le siège est à Bruxelles. Après un mois de prise de connaissance du système et d'hésitations quant au travail à réaliser, nous nous sommes mis d'accord, les dirigeants de la firme, mon promoteur de mémoire et moi-même, pour prendre comme base de travail les tests ET1. Le projet initial était d'implémenter ces tests sur deux machines appartenant à Trasys, mais pour des raisons indépendantes de notre volonté, ces tests n'ont pu être implémenté que sur un seul ordinateur: le Tandem NonStopII.

Au départ, nous ne disposions d'aucun document décrivant précisément le travail que la firme me demandait de réaliser; nous avons donc pris contact avec une personne travaillant à la ville de Bruxelles qui avait mené pour cette administration des tests ET1. Cette personne nous a remis deux articles [1], [2] parus dans la revue Datamation; l'un décrivait le contenu des tests ET1 et l'autre mettait en évidence des éléments permettant d'expliquer les différences entre les résultats obtenus sur différentes machines. Il est apparu par la suite que cet article était d'un intêret quasi-inexistant.

Partant de cela, nous avons cherché à comprendre le contenu des articles, et à dégager une manière d'organiser un plan de travail. Nous avons alors étudié plus en détail les aspects du système Tandem qui seraient utiles pour mener à bien les tests (mesureurs, architecture requester-server, moniteur de transaction, utilitaire de tri,...). Cette recherche nous a amené à prendre contact avec des personnes extérieures, à aller dans des bibliothèques,...

D'après les tests standards décrits dans l'article, nous avons défini les conditions dans lesquelles les tests devaient être menés, les restrictions que nous avions dû porter au standard, les mesures qui devaient être prises et les informations qu'il fallait collecter pour pouvoir interprèter les résultats.

Nous avons mené à bien cette partie du travail, ce qui nous a amené à la fin du mois de janvier. A ce moment est apparu que le travail ne pouvait être transposé sur la seconde machine de Trasys, et nous avons donc décidé

d'adapter un ET1 sur le Vax des facultés. Cette seconde partie a donc été réalisée en parallèle avec les cours et travaux du second semestre.

3. Les Tests ET1: Le standard et ses alternatives.

3.0. Introduction.

Nous allons dans cette partie, définir les 3 tests qui composent le benchmark ET1. Pour cette définition, nous nous sommes basés sur l'article de Datamation [1]. Celui-ci est par ailleurs complété par le rapport technique 85.2 de Tandem [12].

3.1. Présentation générale.

Les tests ET1 ont été conçus par une équipe composée de personnes touchant de près ou de loin à l'informatique. Il y avait des professeurs, des utilisateurs et des vendeurs. L'idée était de définir un standard qui permettrait de comparer le throughput et le rapport prix/performance de différents systèmes de gestion de transaction. Ce standard devait prendre en compte des paramètres oubliés par les mesures prises précédemment, tels que l'architecture I/O du système, son architecture de communication et la puissance du software.

Précisons quelques termes.

On appelle *temps écoulé* le temps nécessaire à une opération pour s'exécuter sur un système vide.

Le *throughput* est défini comme étant le nombre de transactions traitées par seconde par le système.

Selon les cas, la *mesure de performance* sera le temps écoulé ou le throughput.

Le *prix* est calculé comme le coût de l'équipement sur cinq ans, sans tenir compte du coût des lignes de communications, des terminaux et du développement.

Le *rapport prix/performance* est calculé comme le rapport du prix sur la mesure des performances.

Le groupe de travail a défini trois benchmark, chacun d'eux étant destiné à tester un aspect du système..

Une opération de tri. Il s'agit de trier un fichier sur disque, contenant un million d'articles. Les mesures prises sont alors le temps écoulé et le coût.

Une opération de mise à jour séquentielle. Il s'agit d'une transaction COBOL qui lit et met à jour séquentiellement mille articles. Les mesures adéquates sont ici le temps écoulé et le coût.

Une transaction interactive simple. Il s'agit d'une transaction bancaire qui interagit avec un terminal block-mode, connecté via X25. Le système présente un écran de saisie et envoie les données à un programme COBOL qui débite le compte, écrit dans le livre-journal et répond au terminal. La mesure à prendre ici est la valeur du throughput telle que le temps de réponse ne dépasse pas une seconde dans 95 % des cas.

Nous allons étudier plus en détail dans les pages qui suivent chacun de ces tests.

3.2. Le benchmark de tri.

Précisons que les spécifications qui suivent ont été établies par le groupe de travail qui a élaboré les différents benchmarks.

Le fichier à trier est composé d'un million d'articles de cent bytes chacun. Le fichier est organisé de manière séquentielle. Les dix premiers bytes de chaque article consituent la clé de tri. Les clés du fichier d'entrée sont en ordre aléatoire. L'opération consiste à trier ce fichier sur base de la clé.

Les mesures qui conviennent sont le temps écoulé, et le coût.

Ce test est destiné à mettre en évidence les performances entrée/sortie du système, et plus particlulièrement l'architecture hardware I/O.

Le standard ne spécifie pas si l'appel à l'utilitaire se fait directement au niveau du système par un appel au programme Sort ou par l'intermédiaire d'un programme Cobol. Il apparaît que la première solution est plus rapide que la seconde, tout au moins dans le cas du tri de fichiers séquentiels. Quant à l'organisation physique des fichiers (taille des buffers, contiguité ou non,...) , elle est laissée à l'appréciation du programmeur.

3.3. Le benchmark de mise à jour.

Ici encore, les spécifications ont été clairement établies dans le benchmark. Le programme de mise à jour est un programme cobol qui passe en revue séquentiellement un fichier séquentiel, en lisant et en mettant à jour séquentiellement chacun des articles. Le scan global est éclaté en mini-scan,

qui mettent à jour chacun mille articles. On décompose ainsi l'application en 1000 mini-transactions.

L'entrée est constituée d'un fichier séquentiel d'articles de 100 bytes de long dont les dix premiers sont le champ numérique mis à jour. Cette mise-à-jour consiste à augmenter de 5 le montant du champ.

L'algorithme de la transaction est le suivant:

```
OPEN file SHARED,RECORDS LOCKING
PERFORM SCAN 1000 TIMES
BEGIN  -- Start of SCAN TRANSACTION
            BEGIN-TRANSACTION.
            PERFORM 1000 TIMES
            READ file NEXT RECORD record  WITH LOCK
            REWRITE record
            COMMIT-TRANSACTION
END  -- End of SCAN TRANSACTION.
CLOSE FILE
```

Les mesures adéquates sont le temps nécessité par chacune des mini-transactions et le coût.

Ce que l'on veut tester ici, c'est essentiellement les performances I/O software que l'utilisateur final peut tirer du système. Ici encore, il n'y a pas de précisions quant à l'organisation physique des fichiers.

3.4. La transaction débit/crédit.

Cette application laisse une plus grande part à l'interprétation, mais les spécifications qui suivent sont issues de la définition standard des trois benchmarks. Il s'agit d'implémenter une transaction simple.

Une banque compte mille agences, dix mille guichets (dix par agence) et un million de comptes. Elle veut pouvoir effectuer ses transactions on-line.

La base de données de l'application est composée de 4 types de records:

Fichier	Nombre d'articles	Taille du fichier	Taille des articles (Bytes)	Longueur de la clé
Agence	1000	0.1 Mb	100	3
Guichet	10000	1 Mb	100	4
Compte	10000000	1 Gb	100	10
Journal		10 Gb	50	...

Tab. III - 1: Spécification standard des fichiers.

L'algorithme est le suivant:

```
BEGIN-TRANSACTION
            READ MESSAGE FROM TERMINAL (100 b)
            REWRITE  compte
            WRITE  historique
            REWRITE agence
            REWRITE guichet
            WRITE MESSAGE TO TERMINAL (200 b)
COMMIT-TRANSACTION
```

On impose un temps moyen de réflexion au terminal de 100 secondes.

Le standard spécifie encore que le terminal doit être du type block-mode, et que le protocole de communication doit être X25.

En ce qui concerne le choix des données des transactions soumises au système, les numéros d'agence sont générés aléatoirement, ensuite on prend un guichet au hasard dans cette agence. On a ainsi l'origine de la transaction. Il faut enfin le numéro de compte à créditer ou à débiter. Celui-ci est choisi dans 85 % des cas dans l'agence dont le numéro a été généré, et dans 15 % des cas, le compte est choisi dans une autre agence.

Les fichiers doivent être protégés par le blocage des records accédés.

On observe ici l'évolution du temps de traitement d'une transaction, et on s'intèresse plus particulièrement au moment où ce temps dépasse 1 seconde. On pourra ainsi déterminer quel est le throughput tel que le délai de traitement n'excède pas une seconde, dans 95 % des cas.

On voit que, si ces spécifications -issues du standard- sont très précises sur certaines choses, elles le sont moins sur d'autres. Et l'on peut aussi se poser la question de savoir si toutes ces spécifications doivent être respectées scrupuleusement, ou si certaines peuvent être interprètées plus librement. C'est ce que nous allons discuter dans les prochaines pages.

3.5. Interprétations possibles des spécifications ET1.

En ce qui concerne le langage de programmation utilisé pour implémenter la transaction bancaire, le standard prévoit que les programmes soient écrits en Cobol. On peut, il nous semble, prendre cette spécification dans un sens plus large, et dire que la transaction bancaire peut être implémentée dans un langage de haut niveau, permettant une gestion de fichier appropriée.

Le standard prévoit que les terminaux fonctionnent en block-mode, et communiquent avec la machine testée sous le protocole de communication X25. Cependant, le standard ne précise pas le rôle de X25. Il dit juste que c'est dans un but de portabilité du programme. L'esprit du benchmark étant de simuler un environnement bancaire, avec des guichets utilisant de manière

interactive la machine, on peut imaginer d'employer un autre protocole que X25.

Ceci est raisonnable, dans la mesure où de nombreux environnements autres que des terminaux block-mode et X25 peuvent être envisagés. Pensons simplement aux terminaux fonctionnant en character mode, et utilisant le mode de communication DECnet, par exemple.

En ce qui concerne la base de données, on spécifie son contenu (des records de quatre types), mais non son organisation. Il semble que l'on puisse utiliser une base de données relationelle, un gestionnaire de fichiers, ou d'autres systèmes plus sophistiqués.

On pourrait se demander si l'organisation algorithmique de la transaction est correcte: en effet, ne peut-on envisager que la transaction ne comprenne pas les entrées/sorties du terminal? On aurait alors l'algorithme suivant:

```
READ MESSAGE FROM TERMINAL (100 b)
BEGIN-TRANSACTION
              REWRITE  compte
              WRITE  historique
              REWRITE agence
              REWRITE guichet
COMMIT-TRANSACTION
WRITE MESSAGE TO TERMINAL (200 b)
```

Cette structure permettrait d'optimiser le throughput. On peut aussi songer à modifier l'ordre des actions sur la base de données, en fonction de la structure de celle-ci, si cela permet de gagner du temps.

Le standard prévoit que 15 % des transactions se fassent sur un compte non-domicilié dans l'agence traitée. Cependant, si la base de données est organisée de telle manière à ce que cette politique de 85% - 15% ne soit d'aucune influence (par exemple, l'organisation en fichiers séparés, où une agence n'est pas regroupée avec ses comptes et ses guichets), on peut envisager de ne pas implémenter cette politique.

Dans l'article de Datamation[1], on trouve que 95% des transactions doivent être traitées en moins d'une seconde. Mais cette contrainte peut dépendre de l'architecture du système. Il est possible de modifier cette contrainte en disant que 90% des transactions devaient être traitées en moins de 2 secondes. Ceci a été pratiqué dans différents cas [3], [4].

En ce qui concerne le blocage des records dans la base de données, le standard prévoit qu'au moins le blocage de record soit implémenté. Il laisse au programmeur le soin de déterminer le degré de granularité qu'il veut fixer. D'autres choix sont encore possibles, comme le blocage d'un bloc, d'une page,... Mais il faut tenir compte de l'impact qu'à le degré de bloqueage sur la concurrence d'accès aux articles. Plus la concurrence est grande, moins le throughput sera élevé.

Pour ce qui est de la taille de la base de données, on peut penser que plus elle est réduite, plus les temps d'accès seront brefs. Cependant, il faut aussi prendre en considération le fait que si l'on augmente sa taille, on diminue le risque de concurrence d'accès à un même article, ce qui réduit d'autant le risque d'attente d'un article bloqué, et augmente d'autant le throughput du système.

On voit ainsi que diverses interprétations peuvent être données à certaines des spécifications du standard, sans pour autant porter atteinte à l'esprit initial du benchmark.

Nous allons maintenant donner quelques éléments qu'il faudrait prendre en compte lorsque l'on se trouve face à des résultats d'ET1 menés sur des machines différentes, pour interprèter correctement les résultat

3.6. Critères de comparaisons d'ET1.

Une première remarque concerne les tests réellement implémentés. En effet, la plupart des benchmarks ET1 se contentent de la transaction bancaire,

en laissant de côté les deux autres opérations. Il ne faut pas oublier que chacun des trois tests est destiné à tester un certain aspect du système, et que pour avoir une idée générale des performances du système, il convient d'utiliser les trois tests.

Il faut aussi clairement définir sur quelle configuration hardware les tests ont été menés. Le benchmark original a été mené sur une machine Tandem; pour pouvoir comparer les résultats des tests menés, il faut spécifier correctement la configuration hardware du système afin de ramener les résultats sur une même échelle de comparaison.

Un autre paramètre important est la manière dont les utilisateurs sont simulés. Le benchmark original simulait les utilisateurs par des terminaux travaillant en block-mode et communiquant via des lignes de communication suivant le protocole X25. Pour pouvoir comparer les résultats de tests ET1, il faut que l'overhead engendré par le système de communication puisse être correctement isolé, car selon le type de simulation des terminaux et de communication, on aura un overhead différent, et par là, des résultats non-comparables.. Dans notre cas, les terminaux étaient des terminaux block-mode, mais les lignes de communication suivaient le protocole SNA pour ce qui est de la machine Tandem. Pour ce qui est du Vax, les terminaux travaillaient en character-mode.

Il faut aussi tenir compte du taux de transaction utilisé. Le standard prescrivait un think time de 100 secondes; il est important de voir si le benchmark implémenté utilise ce think time, un autre, un think time variable,...En effet, le think time va influencer considérablement le temps de réponse; plus le think time est grand, plus le temps de réponse sera bon, pour autant que le nombre d'utilisateurs reste constant. Dans notre cas, nous avons dû faire varier le temps de réflexion pour pouvoir faire croître le nombre de transactions traitées par seconde.

On peut se demander quelles sont les mesures prises lors de l'exécution du benchmark et comment elles ont été prises ? Le benchmark original produisait les sorties suivantes: transactions par seconde, coût par transaction et temps écoulé. Pour pouvoir comparer deux benchmarks ET1, il faut évidemment qu'ils produisent des sorties comparables. Dans notre cas, nous

avons pris le temps écoulé et le nombre de transactions par seconde; le calcul du coût faisant intervenir des facteurs propres à l'entreprise et plutôt confidentiels, il a été laissé de côté.

L'utilisation d'une base de données de taille réduite par rapport à celle du standard peut fausser les résultats obtenus; si la base de données est de taille fort réduite par rapport à la taille standard, les fichiers peuvent tenir en mémoire cache, ou, en tout cas, provoquer moins de page-faults et par là-même améliorer les performances en réduisant le nombre d'entrées-sorties. Dans notre cas, la taille des fichiers a dû être modifiée pour des questions de place-disque disponible.

Notons que la structure des fichiers peut être optimisée pour améliorer les performances du benchmark. Dans le standard et dans notre cas, la structure des fichiers est fixée par le système, et excepté certains paramètres mineurs, ne pouvait être modifiée pour optimisation par l'utilisateur. Cependant, on peut imaginer l'incidence d'un buffer de lecture sur disque de taille importante, ou du nombre de niveaux d'index sur les temps d'accès à un article.

Dans le même ordre d'idée, il faut voir si le benchmark implémenté contient un montant important de données redondantes,ce qui permettrait à certains moments d'éviter l'accès à certains fichiers? En effet, une telle redondance permettrait de réduire les temps d'accès aux articles.

Pour ce qui concerne le type de blocage/protection implémenté, le standard prévoit le blocage de l'article auquel on accède pour le temps de l'opération et uniquement de cet article-là. C'est ce qui a été implémenté dans notre benchmark. Il faut remarquer que le type de blocage peut influencer le temps d'accès aux articles, et donc le temps de réponse.

Le standard prévoit que toutes les mises à jour faites aux fichiers doivent être écrites dans un journal, pour permettre une reprise sur erreur. Si l'on n'implémente pas cette écriture au journal, cela améliorera les performances du benchmark, mais on s'éloigne du standard et des applications OLTP réelles, dont la plupart imposent la tenue d'un journal. Dans le cas du système Tandem, un journal était gardé à jour automatiquement par le système; par contre, dans le système Vax, un tel système de logging n'était pas implémenté.

Enfin, un facteur important est le choix des données des transactions soumises au système. Il est crucial que les données soient choisies de manière aléatoires, car si ce n'est pas le cas, on risque de travailler sur la même zone de données, chargée une fois pour toute en mémoire centrale, et de ce fait éliminer de nombreux accès disque, en faussant ainsi les performances d'un système qui n'est plus représentatif d'une application 'monde-réel'.

4. Présentation du système Tandem NonStopII et des outils disponibles.

4.1. Description du système Tandem Non Stop II.

4.1.1. Organisation Hardware.

L'ordinateur est organisé autour de deux processeurs TXP, fonctionnant en parallèle et indépendemment, avec une vitesse d'exécution de 2 Mips par processeur. La longueur des mots est de 16 bits, mais la longueur d'adressage est de 32 bits. Chaque processeur dispose d'une mémoire cache de 4096 pages, soit 8 Kb(une page a une taille de 2 K). Les processeurs peuvent communiquer entre eux via un bus interprocesseur dont le débit est de 13.33 Mb/sec.

Le module processeur a la structure suivante:

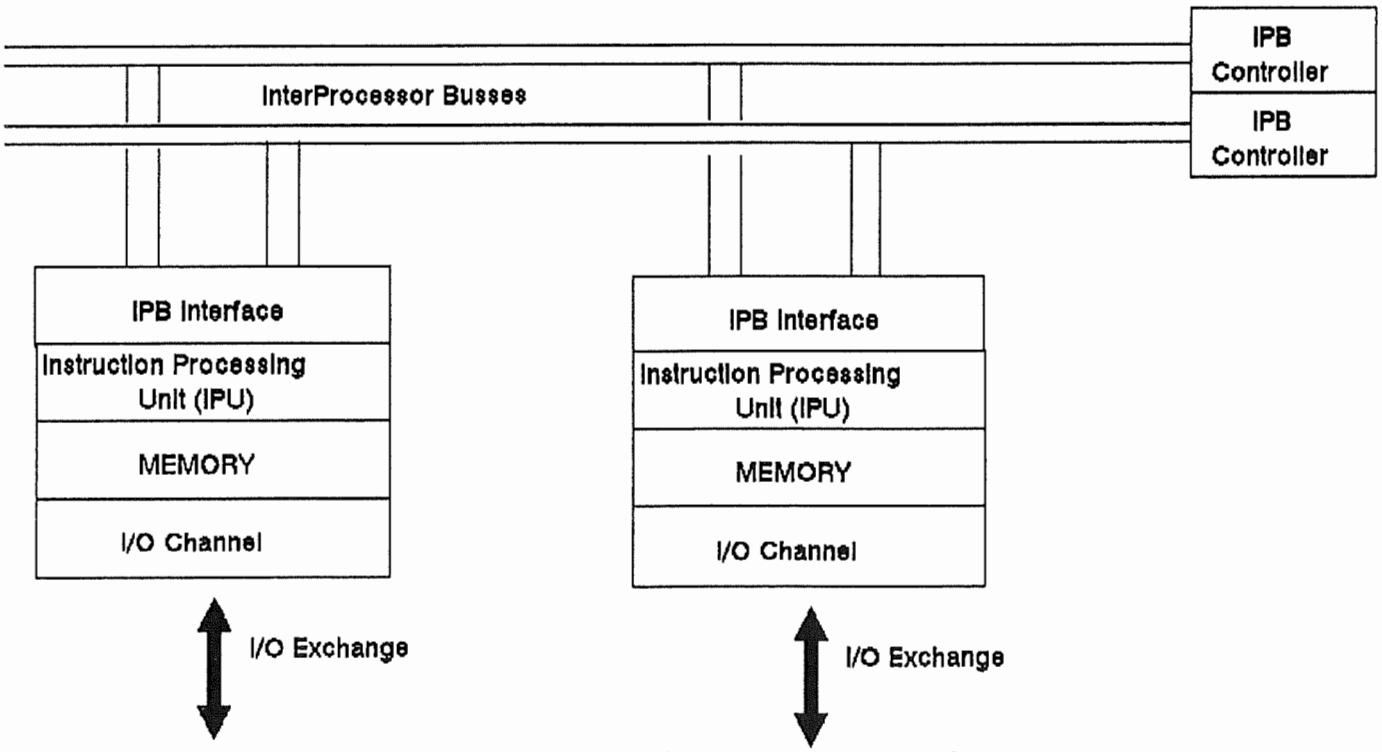


Fig. IV - 1: Structure des processeurs sur Tandem.

Le module processeur est composé de différents éléments:

* Une interface gérant les échanges avec le bus interprocesseur (IPB · InterProcessor Bus)

* L'unité de traitement des instructions (IPU·Instruction Processing Unit) qui en plus d'exécuter les instructions·machines, gère aussi le transfert de contrôle de l'IPU lors d'interruptions, et enfin gère le transfert des données de l'IPB vers la mémoire.

* La mémoire

* Un module de gestion des échanges avec les dispositifs d'entrée/sortie.

Les disques sont au nombre de quatre. Chaque disque a une capacité de 400 Mb et dispose d'un disque miroir. Le seek time moyen est de 15 ms, le délai rotationnel est de 9.77 ms, et la vitesse de transfert est de 1.86 MB/sec

La particularité du système Tandem est qu'il est organisé pour ne jamais tomber en panne. Ceci est réalisé grâce aux caractéristiques suivantes:

* Il y a deux processeurs indépendants.

* Les bus de communications interprocesseurs sont dédoublés.

* Chaque contrôleur I/O peut être géré par l'un ou l'autre processeur.

* Chaque disque est géré par deux contrôleurs.

* Il y a plusieurs sources d'alimentation.

* Une reprise sur panne d'alimentation est réalisée.

La redondance dans les canaux de communication vise à garantir un maximum de sécurité à l'utilisateur du système. Par exemple, si un processeur tombe en panne, sa charge est automatiquement répartie sur le(s) autre(s) processeur(s). Il en est de même pour les contrôleurs de disques. En effet, chacun de ces contrôleurs est géré par un processeur défini comme primaire, mais si ce processeur primaire tombe en panne, c'est le processeur défini comme secondaire qui prend la relève. Etant donné le dédoublement des connections entre les processeurs et les disques, les chemins d'accès sont ainsi multipliés. Rappelons aussi que chaque disque dispose d'un disque miroir, c'est-à-dire que toute modification (écriture, modification ou effacement d'un article) apportée à un fichier sur le disque primaire est répercutée automatiquement sur le disque miroir. De plus, cette seconde écriture sur disque se fait par un chemin d'accès différent du premier, ceci afin de garantir une fiabilité maximum au système, et réaliser une économie de temps.

Voici le schéma de l'architecture de l'ordinateur Tandem NonStop; les traits pleins représentent les liens primaires entre les appareils alors que les traits en pointillés représentent les liens secondaires.

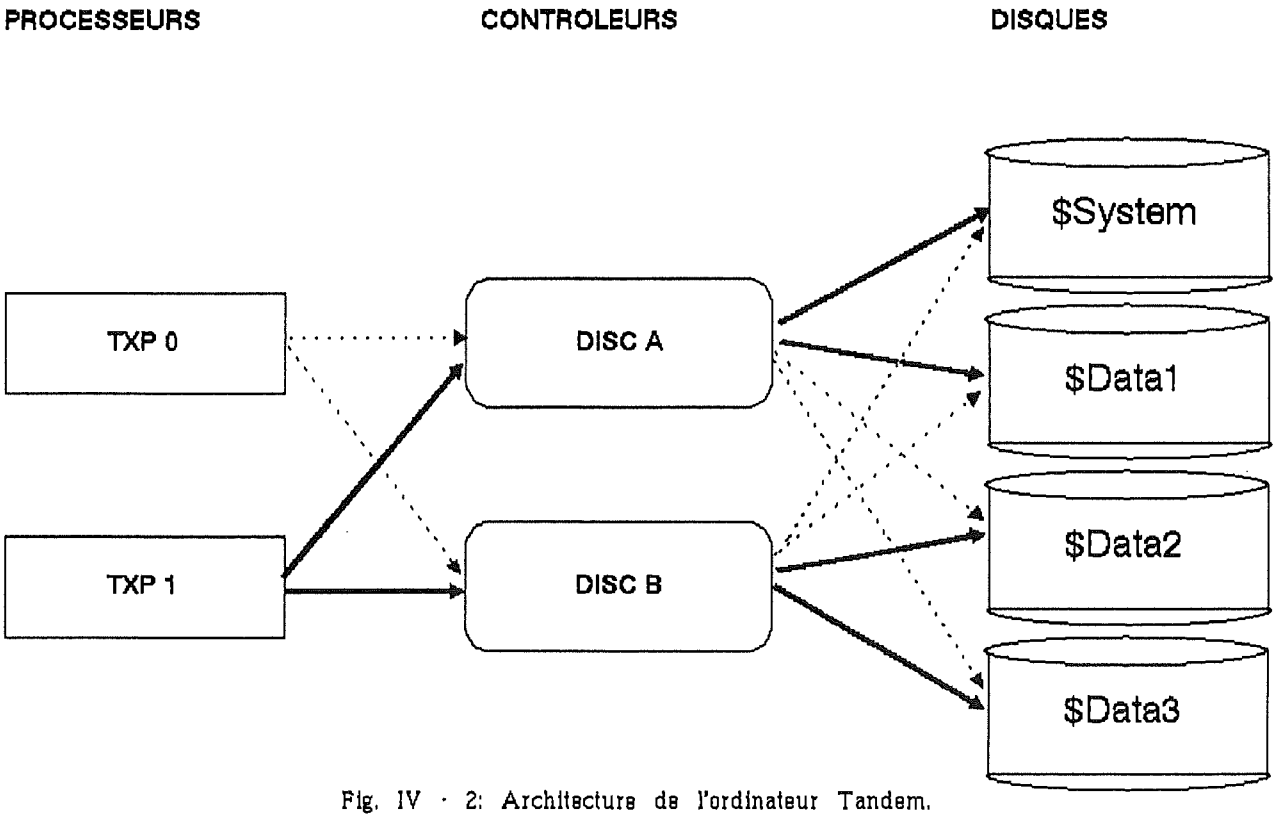


Fig. IV - 2: Architecture de l'ordinateur Tandem.

On voit sur ce schéma que le processeur primaire pour chacun des contrôleurs est le processeur 1.

L'ordinateur se trouve à Kallo, près d'Anvers, et les bureaux de Bruxelles y sont reliés par une liaison SNA bitsynchrone à 14.400 bauds.

4.1.2. Organisation software.

Les tâches confiées au processeur, s'appellent processus. Chacun de ces processus est caractérisé par un nom, un numéro identifiant, un propriétaire et une priorité.

Les processus sont répartis sur les deux processeurs, de manière à équilibrer la charge du système. Un processus peut être lancé sur les deux processeurs en même temps; il faut alors définir quel est le processeur primaire et quel est le processeur de backup. Ceci permet, au cas où le processeur primaire aurait une défaillance, de poursuivre l'exécution du processus interrompu sur le processeur de backup.

Le système d'exploitation est le OS·Guardian. L'interpréteur de commande s'appelle TACL .

Ce système est essentiellement destiné à une gestion de transactions. Néanmoins, il est possible d'écrire des programmes en différents langages (Cobol, TAL), de les compiler et de les exécuter directement.

Pour bâtir un système de gestion de transaction, la tâche est plus ardue; en effet, l'architecture Tandem prévoit deux sortes de programmes: les servers et les requesters. Les premiers sont destinés à gérer une base de données, tandis que les seconds sont chargés de la gestion du(des) terminal(aux) de l'application. Ainsi, pour écrire une application, on écrit un ou plusieurs requesters dans un langage appelé SCOBOL (pour **S**creen **C**obol), qui définit les formats d'écran, la structure des données,... Ensuite, il faut écrire les programmes qui gèreront la base de données: les servers. Ceux-ci sont écrits en Cobol.

Le lien entre ces deux types de programmes est réalisé par un pseudo-fichier, appelé $RECEIVE, et dans lequel chaque programme peut lire ou écrire.

Le schéma-type d'une application est le suivant:

Requester	**Server**
Lire les données à l'écran **L**es envoyer au server concerné par une écriture dans le fichier $Receive	
	Lire un message dans le fichier $Receive **Me**ttre à jour la Base de données d'après les données reçues. **E**nvoyer un message de réponse par une écriture dans le fichier $Receive
Lire le message reçu du serveur dans le fichier $Receive.	

Les deux programmes tournent indépendemment l'un de l'autre. Le server reste toujours en attente d'un message, jusqu'à ce qu'un ordre spécifique l'arrête.

Plusieurs requesters peuvent utiliser le même server. De même, un requester peut appeler plusieurs servers.

Pour éviter une surcharge d'un server, un dispositif de gestion dynamique est implémenté. Si au bout d'un délai préfixé, une requête arrivée à un serveur n'est toujours pas traitée, le système génère automatiquement une copie supplémentaire de ce server. Inversément, si au bout d'un délai préfixé, une copie de server n'a plus reçu de requêtes à traiter , elle est automatiquement détruite par le système.

4.2. Outils disponibles.

Le système offre différents outils à l'utilisateur: outre un éditeur de texte conventionnel, on trouve des compilateurs COBOL, SCOBOL et TAL. On dispose aussi d'un mesureur (MEASURE) permettant de prendre les mesures suivantes:

Processeur

Temps d'occupation du processeur
Temps passé par les processus en état READY (Un processus est READY s'il est en train de touner, s'il est en attente de la gestion d'un page-fault ou s'il a été sélectionné dans la liste d'attente.)
Nombre de swaps réalisés
Nombre de dispatch

Processus

Temps passé dans le CPU
Temps d'attente de la gestion d'un page-fault.
Nombre de page-faults
Temps passé en état READY
Nombre de fois que le processus a été sélectionné et exécuté (DISPATCH).

Accès logiques à un fichier

Temps d'exécution des I/O requests
Nombre d'appel à une procédure de READ
Nombre d'appel à une procédure de WRITE
Nombre d'appel à une procédure de UPDATE

Accès physiques à un fichier

Nombre de READ qui ont causé une entrée/sortie disque
Nombre de WRITE qui ont causé une entrée/sortie disque
Nombre de fois qu'un READ a trouvé le bloc cherché en mémoire cache
Nombre de fois qu'un WRITE a trouvé le bloc cherché en mémoire cache

Disque

Temps d'occupation du disque
Temps d'attente des requêtes
Nombre de I/O requests reçus
Temps passé à faire des READ
Temps passé à faire des WRITE
Temps passé à faire des SEEK
Nombre de READ
Nombre de WRITE

Nombre de SEEK
Nombre de bytes lus sur le disque
Nombre de bytes écrits sur le disque
Nombre de fois que le bloc cherché se trouve en cache
Nombre de fois que le bloc cherché ne se trouve pas en cache
Nombre de fois que le bloc cherché devrait être en cache et n'y est pas suite à une opération du gérant de la mémoire.

TMF (Transaction Monitoring Facility).

Nombre de transactions lancées dans un CPU particulier
Temps d'existence de transactions lancées dans ce CPU
Nombre de transactions qui ont échoué
Temps d'attente d'une transaction qui a échoué pour le back-out

4.3. Le générateur de transactions (ENCORE)

Ce logiciel permet d'enregistrer les interactions lancées depuis un terminal, de les modifier, et puis de les rejouer en simulant l'activité de plusieurs terminaux simultanément.

Ceci est fait en interposant un processus entre le terminal et le processus qui le gère (TCP). Ce processus enregistre les interactions entre le terminal et son TCP et sauve le tout dans un fichier. Ce fichier peut alors être édité pour modifier les données introduites, ou le temps de réflexion de l'opérateur qui les a introduites. ENCORE permet alors de rejouer le scénario enregistré, mais en le multipliant, pour simuler l'activité de plusieurs terminaux. La multiplication peut se faire soit sur le temps de réflexion(qui peut être multiplié par 100 au plus), soit sur le nombre de terminaux (ce nombre ne peut excéder 12).

4.4. Principes de fonctionnement de l'utilitaire de tri (FAST-SORT).

L'algorithme de tri implémenté sur la machine Tandem tire parti des caractéristiques de cette machine, entre autre le traitement parallèle. Il permet que le temps nécessaire au tri d'un fichier varie de manière approximativement linéaire avec la taille de ce fichier, pour autant que cette taille soit suffisament importante.

Le procédé travaille en une ou plusieurs passes, selon la taille du fichier à trier. Durant la première passe, FASTSORT lit les records d'entrée et les range dans un arbre binaire arrangé de la manière suivante: au départ, l'arbre est rempli de records nuls maximums. FASTSORT ajoute les records aux feuilles en déplaçant les records nuls vers la racine. Le record maximum se trouve à la racine de l'arbre, et le 'gagnant' d'un sous-arbre est la racine de ce sous-arbre. Cette technique de l'arbre minimise le nombre de fois que FASTSORT compare un record avec un autre; un arbre de hauteur 14 peut contenir 16385 records (2E14+1), et FASTSORT compare chaque record avec 14 autres quand il trie tout l'arbre. Si un record a une longueur de 100 bytes, un tel arbre occupe approximativement 16385*100, soit environ 1.7 Mbytes.

Les records sont lus dans des blocs de 6Kb, pour minimiser le nombre d'accès au(x) disque(s). De plus, ces buffers de lecture et d'écriture sont dédoublés, c'est-à-dire que FASTSORT, en même temps qu'il trie un bloc, va déjà chercher le bloc suivant. De même, pour la sortie, FASTSORT écrit un bloc sur le disque en même temps qu'il charge remplit le bloc suivant en mémoire.

Le tri peut se faire en une passe si le fichier entier peut tenir en mémoire centrale. Les records sont lus et ajoutés aux feuilles de l'arbre. Une fois que tous les records ont été lus, ils sont complètement triés dans l'arbre qui est donc prêt à être recopié dans un fichier de sortie. La mémoire centrale allouée à un processeur peut être au plus de 16 MBytes. Mais sur un système multi-processeurs, le tri global est divisé en tris moins importants (subsorts) distribués dans les différents processeurs. Chacun d'eux fait donc une partie du travail et renvoie le résultat à un collecteur qui met ensemble les sous-ensembles triés.

On peut remarquer que quand durant la première passe du tri, les accès au(x) disque(s) sont purement séquentiels; donc le tri parallèle est limité par la vitesse d'accès au disque.

Cette solution en une passe, même si elle est la plus rapide, n'est pas nécessairement la moins chère. Il faudrait envisager un algorithme à plusieurs passes pour les fichiers de grande taille, dans une approche moins memory-intensive.

Si le fichier est plus grand que l'arbre, on va procéder comme dans la première méthode; on va lire les records du fichier d'entrée et les ranger dans la structure d'arbre. Quand l'arbre est plein, deux alternatives sont possibles:

* le nouveau record est plus grand que la racine de l'arbre. Dans ce cas, on sauve l'arbre dans un scratch-file, et on recommence un autre arbre avec le nouveau record.

* le nouveau record est plus petit que la racine de l'arbre. Dans ce cas, on sauve la racine de l'arbre dans un scratch-file, ce qui résulte en un trou dans l'arbre. Ce trou est bouché par le nouvel article, et l'arbre est recalculé.

Ce processus donne donc différents scratch-files déjà triés. La moyenne de la longueur des séries a été établie statistiquement par Knuth [14]: elle vaut deux fois la taille de l'arbre.

Remarquons que si le fichier arrive presque trié, une seule série sera générée, alors que si le fichier est trié dans l'ordre inverse (ce qui est le pire des cas), une série aura comme longueur le nombre de records pouvant être mis dans l'arbre. D'après la description fournie du procédé , il apparaît que chacune de séries générées est stockée dans un scratch-file différent.

Quand plusieurs séries sont générées, FASTSORT doit encore les fusionner. Cette fusion se fait encore à l'aide d'un arbre de tri. Cependant, cette fusion est donc limitée dans le nombre de séries en entrée par la taille de l'arbre de tri. Si la première phase a généré 14 séries, et que la taille de l'arbre est 10, on ne peut fusionner que 10 séries à la fois, au maximum. On fusionnera donc 5 séries, pour commencer, et on refusionnera la série générée avec les 9 séries restantes.

5. Présentation du système VAX/VMS et des outils disponibles.

5.1. Description du système VAX/VMS.

5.1.1. Organisation hardware.

L'ordinateur est organisé autour de deux processeurs CMOS, de 2.8 Vup chacun. Il est très difficile de convertir cela en Mips, mais d'après les responsables du centre de calcul, cela devrait correspondre à 1 Mips ou 1.5 Mips, selon que l'on travaille ou non en floating point. Les deux processeurs travaillent en symetric processing. A tout moment, une tâche peut être prise en charge par l'un ou l'autre des processeurs, indifféremment. Chaque processeur dispose d'une mémoire centrale de 64 Mbytes. La longueur des mots est de 32 bits.

Les disques sont au nombre de 10. Ce sont des disques de type RA81, de 450 Mbytes chacun. Le taux de transfert est de 2.20 MB/Sec, le seek time moyen est de 28.0 ms et le délai rotationnel est de 8.3 ms.

Le chemin d'un terminal au système central est le suivant:

Chemin de communication du terminal à l'ordinateur.

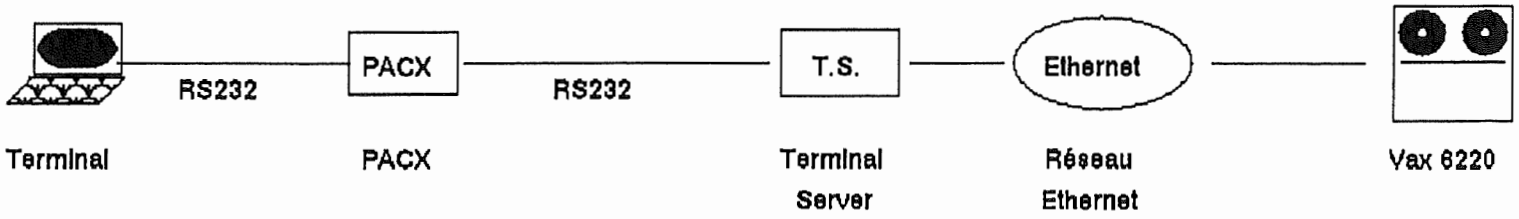


Fig. V - 1

5.1.2. Organisation software.

Les tâches confiées aux processeurs sont appelées processus. Chacun d'eux est identifié par un numéro (PIN), un nom (facultatif) et une priorité.

Les tâches sont placées dans une file d'attente, et elles peuvent être prises en charge par l'un ou l'autre des processeurs.

Le système d'exploitation est Vax/Vms et l'interprèteur de commande s'appelle DCL.

Il s'agit d'un general-purpose system. En effet, il n'est pas dédicacé à un usage particulier, comme c'était le cas chez Tandem.

On peut développer des applications en Cobol, Pascal, Basic, Assembler. On peut manipuler des bases de données, soit de type Codasyl avec DBMS ou relationnelle avec RDB/SQL.

On ne note pas sur le Vax, la présence d'une architecture Requester-Server. Les fonctions de gestion des écrans et de manipulation de bases de données ou fichiers, sont réalisées par un seul et même programme.

Nos applications ayant été développées en Cobol, nous avons utilisé des fichiers de type séquentiel et indexé. Il serait intéressant de dire un mot de l'organisation de ces fichiers sur le Vax.

Pour les fichiers séquentiels, les articles sont stockés de manière telle que le suivant physique d'un article est aussi son suivant logique. Les articles sont rangés dans des blocks dont la taille est 512 Kbytes. Quant à la contiguité, elle n'est pas assurée. En effet, à la création d'un fichier, on peut spécifier quelques paramètres, dont une option de contiguité. Mais comme les fichiers sont créés à partir d'un programme Cobol, ces options n'ont pas pu être activées.

Dans le cas des fichiers indexés, les articles sont rangés dans des buckets. Un bucket est un ensemble de blocks, utilisé comme structure de stockage pour des fichiers indexés et comme unité de transfert vis-à-vis du disque. Le

nombre de blocks par bucket est défini globalement pour un fichier. Les articles sont rangés dans les buckets triés sur la clé primaire. Pour un bucket, la valeur de clé maximum est celle du dernier article, et c'est cette valeur qui est utilisée pour bâtir l'index.

Dans les deux cas (séquentiel ou indexé), lorsque l'on crée un fichier et qu'on le garnit, on écrit le nouvel article dans le block courant, s'il n'est pas plein, et s'il est plein, on écrit le bloc sur disque, et on en utilise un nouveau. Cela est l'option par défaut, car parmi les paramètres spécifiables, on trouve aussi le taux de remplissage des blocks ou des buckets. Ce taux doit être compris entre 50 et 100%. A ce moment, les blocks ou buckets seront remplis en fonction de ce taux.

Remarquons que si on a une insertion à réaliser, dans le cadre d'un fichier indexé, et que l'insertion doit se faire dans un bucket plein, on crée un nouveau bucket et on laisse dans le premier la moitié des articles qui le composaient, et l'autre moitié des articles plus le nouveau va dans le nouveau bucket. C'est ce qu'on appelle le bucket-split. Et c'est pour éviter ce phénomène que l'on permet de déterminer à l'avance le taux de remplissage des buckets ou des blocks.

Notons encore une fonctionnalité du système; il permet de comprimer les données stockées, dans le cas de fichiers séquentiels, et ce, à trois niveaux:

* Key compression: Ce sont les valeurs de clés dans les buckets de données qui sont comprimées.

* Index compression: ce sont les valeurs de clés dans les buckets d'index qui sont comprimées.

* Data compression: ce sont les valeurs de données elles-mêmes qui sont comprimées.

Toutes ces fonctionnalités sont fournies pour permettre de gagner de l'espace disque, ou de raccourcir le temps d'accès aux articles.

5.2. Outils disponibles.

Pour permettre de développer des programmes, le système offre un éditeur de texte, différents compilateurs, et des primitives d'appels-système, permettant d'obtenir certains renseignements ou certaines fonctions.

On ne trouve pas, sur le Vax, de mesureur comme celui disponible sur Tandem. On ne peut spécifier les entités à mesurer, ou les mesures exactes à prendre. De plus, pour pouvoir l'utiliser, il faut compiler le mesureur avec le programme à mesurer, les linker ensemble, pour les exécuter, ce qui modifie le temps d'exécution. Par ailleurs, les résultats fournis par ce programme constituent plus une analyse des parties de texte de programme fort ou peu exécutées (ce qui fournit des indications pour une optimisation éventuelle du programme), plutôt qu'au niveau des ressources véritablement consommées.

De plus, comme la mesure qui nous intéressait était le temps écoulé, elle pouvait être prise par de simples appels-système.

Sur le système Vax, il n'y a pas non plus de générateurs de transactions, comme cela existait sur le Tandem. Dès lors pour simuler l'activité de différents terminaux, il a fallu s'y prendre autrement.

5.3. Principes de fonctionnement de l'utilisaire de tri.

L'algorithme de tri est assez similaire à celui du Tandem.

Le procédé travaille en une ou plusieurs passes, selon la taille du fichier à trier. Durant la première passe, le programme lit les records d'entrée et les range dans un arbre binaire arrangé de la manière suivante: au départ, l'arbre est rempli de records nuls maximums. Le processus de tri ajoute les records aux feuilles en déplaçant les records nuls vers la racine. Le record maximum se trouve à la racine de l'arbre, et le 'gagnant' d'un sous-arbre est la racine de ce sous-arbre. Cette technique de l'arbre minimise le nombre de fois que l'on compare un record avec un autre.

Les records sont lus dans des blocs de 8192 bytes (taille du bloc de transfert entre le disque et la mémoire, pour un fichier séquentiel).

Le tri peut se faire en une passe si le fichier entier peut tenir en mémoire centrale. Les records sont lus et ajoutés aux feuilles de l'arbre. Une fois que tous les records ont été lus, ils sont complètement triés dans l'arbre qui est donc prêt à être recopié dans un fichier de sortie. La mémoire centrale allouée au processus de tri est de 10000 pages.

On peut remarquer que, comme dans le cas du Tandem, durant la première passe du tri, les accès au disque sont purement séquentiels.

Cette solution en une passe, même si elle est la plus rapide, n'est pas nécessairement la moins chère. Il faudrait envisager un algorithme à plusieurs passes pour les fichiers de grande taille, dans une approche moins memory-intensive.

Si le fichier est plus grand que l'arbre, on va procéder comme dans la première méthode; on va lire les records du fichier d'entrée et les ranger dans la structure d'arbre. Quand l'arbre est plein, deux alternatives sont possibles:

* le nouveau record est plus grand que la racine de l'arbre. Dans ce cas, on sauve l'arbre dans un scratch-file, et on recommence un autre arbre avec le nouveau record.

* le nouveau record est plus petit que la racine de l'arbre. Dans ce cas, on sauve la racine de l'arbre dans un scratch-file, ce qui résulte en un trou dans l'arbre. Ce trou est bouché par le nouvel article, et l'arbre est recalculé.

Ce processus donne donc différents scratch-files déjà triés. La moyenne de la longueur des séries a été établie statistiquement par Knuth [14]: elle vaut deux fois la taille de l'arbre.

Remarquons que si le fichier arrive presque trié, une seule série sera générée, alors que si le fichier est trié dans l'ordre inverse (ce qui est le pire des cas), une série aura comme longueur le nombre de records pouvant être mis dans l'arbre. D'après la description fournie du procédé , il apparaît que chacune de séries générées est stockée dans un scratch-file différent.

Quand plusieurs séries sont générées, il faut encore les fusionner. Cette fusion se fait encore à l'aide d'un arbre de tri.

6. Implémentation des tests ET1 sur le Tandem choix et restrictions.

6.0. Introduction.

Dans la suite, je vais décrire, pour chaque test, la manière dont il a été implémenté sur l'ordinateur Tandem, ainsi que les choix, hypothèses et restrictions que cette implémentation a entraînés.

Un premier choix dans l'implémentation est l'utilisation du langage cobol. Mais étant donné que c'est le seul langage évolué disponible, le choix s'imposait.

De plus, j'avais un disque de 400 Mb à ma disposition. Il fallait donc faire tenir tous mes fichiers dessus. L'organisation des fichiers sur le Tandem est telle qu'il faut créer explicitement chaque nouveau fichier, en définissant sa taille maximum, la taille du buffer de lecture (ou taille des blocs) qui doit être un multiple de 512 bytes et son type (séquentiel, indexé, relatif, non-structuré).

A la création, les articles sont rangés dans les blocs. On met le plus grand nombre entier possible d'articles dans un bloc, ce qui donne un taux de remplissage très élevé. Ceci est imposé par le système Tandem.

Les blocs sont rangés dans des pages physiques, dont la taille est 2K. Dans la mesure du possible, ils sont rangés de manière contigüe, c'est à dire qu'entre le premier bloc du fichier et le dernier, il n'y a pas de bloc libre. La taille des pages est fixée lors de la génération du système.

6.1. Le test Sort.

J'ai décidé de lancer l'opération de tri sur des fichiers de tailles différentes; d'abord un million d'articles, comme prévu dans le standard, ensuite, cent mille, dix mille, mille et cent articles.

Chaque article a une longueur de cent bytes, dont les dix premiers constituent la clé de tri, comme prévu dans l'article [1].

Les articles sont rangés dans des blocs de 1024 bytes, à raison de 9 articles par bloc. Les blocs sont remplis en moyenne à 92%. Le fichier est stocké de manière contigüe. Ceci est dû à l'organisation des fichiers du Tandem.

Le chargement de ce fichier est réalisé par un programme cobol (BDSORT), dont le texte est joint en annexe, qui garnit les valeurs de clé à l'aide d'un générateur de nombres aléatoires.

L'opération de tri proprement dite est lancée elle aussi depuis un programme cobol (ET1SORT), dont le texte est en annexe, qui consiste en un appel à l'utilitaire de tri décrit plus haut.

On voit que il n'y a pas eu de choix à faire, ni de restriction à imposer au standard.

6.2. Le test Scan.

Comme pour le test Sort, j'ai fait tourner le Scan sur des fichiers de tailles différentes: un million, cent mille, dix mille, mille et cent articles.

La structure du fichier d'entrée est la suivante: Chaque article a une longueur de cent bytes dont les dix premiers constituent un champ numérique mis à jour par l'opération de Scan, comlme prévu dans le standard [1].

Les articles sont rangés dans des blocs de 1024 bytes, à raison de 9 articles par bloc. Les blocs sont remplis en moyenne à 92%. Le fichier est stocké de manière contigüe. Ceci est dû à l'organisaion des fichiers sur Tandem.

Le chargement de ce fichier est réalisé par le même programme que pour le Sort (BDSORT).

La procédure de Scan est réalisée par le programme cobol ET1SCAN, dont le listing se trouve en annexe. Il parcourt séquentiellement les articles et les met à jour, comme décrit dans le standard. Une transaction est constituée par la mise à jour d'un groupe d'articles.

Ici encore, à part le choix de faire varier la taille du fichier, il n'y a pas eu d'autre décision à prendre.

6.3. La transaction bancaire.

Ici, j'ai dû modifier le standard; en effet, je ne disposais pas sur le disque de la place physique pour stocker les fichiers de la taille décrite dans l'article [1]. Voici les tailles telles que réellement implémentées.

Fichier	Nombre d'articles	Taille du fichier	Taille des articles (Bytes)	Longueur de la clé (en bytes)
Agence	1000	0.1 Mb	100	3
Guichet	10000	1 Mb	100	4
Compte	1000000	100 Mb	100	10
Journal	200000	10 Mb	50	...

Tab. VI - 1: Spécification des fichiers sur Tandem.

Les fichiers sont indépendants. Les trois premiers -respectivement les fichiers agence, guichet et compte- sont organisés de manière indexée, sur la clé composée par -respectivement- les 3,4 et 10 premiers bytes de l'article comme prescrit par le standard. Les articles de ces trois fichiers sont rangés dans des blocs de 512 bytes, rangés de manière contigüe. Le fichier agence compte deux niveaux d'index, avec un taux de remplissage au niveau des données de 86%. Le fichier guichet compte lui aussi deux niveaux d'index, avec un taux de remplissage au niveau des données de 86%. Le fichier Compte nécessite quatre niveaux d'index et présente un taux de remplissage de 86%

lui aussi. Tout ceci est le résultat de l'organisation des fichiers sur Tandem. Les trois fichiers (agence, guichet, client) ne subissent que des mises à jour de champs d'articles existants; le taux de remplissage ne varie donc pas avec l'utilisation.

Le fichier journal est organisé de manière séquentielle. Les articles, d'une longueur de 50 bytes, sont rangés dans des blocs de 1024 bytes; on a donc 19 articles par blocs, et un taux de remplissage de 99%. Ce fichier ne subit que des écritures de nouveaux articles. Le principe de remplissage est le suivant: on écrit le nouvel article à la suite du dernier écrit, en s'arrangeant pour que un article soit stocké physiquement après l'article logique le précédant.

On trouvera la description détaillée des fichiers en annexe.

Ces fichiers sont chargés par des programmes cobol dont le texte se trouve en annexe. Les articles des fichiers indexés sont créés et écrits séquentiellement, avec une valeur de clé attribuée par compostage.

Le fichier journal, quant à lui, est chargé par une écriture séquentielle d'articles de 50 bytes de long.

La transaction débit/crédit est réalisée par un requester et un server, nommés ET1REQ et ET1SERV, dont les listings se trouvent en annexe.

Le requester suit l'algorithme suivant:

```
init.

loop until end.
        Display screen
        Accept data from terminal
        Send data to server
end-loop.
```

Il exécute la boucle jusqu'à un message d'arrêt venant du terminal.

Le server suit l'algorithme suivant:

```
init.

loop until stop
        read data from requester
        update compte
        update guichet
        update agence
        write journal
        send message to requester
end·loop.
```

Il exécute la boucle jusqu'à ce qu'un message du système lui demande de s'arrêter.

J'ai utilisé le logiciel ENCORE (voir description ci·dessus) pour enregistrer une série d'interactions. Ensuite, j'ai modifié le temps de réflexion de chacune d'elle en le mettant à 100 sec., pour rester conforme au standard. Cependant, comme le générateur de terminaux ne permettait pas de simuler l'activité de plus de 12 terminaux, il a fallu que je réduise le temps de réflexion pour faire augmenter le throughput et avoir des résultats significatifs.

Voyant que les transactions mettaient en général plus d'une seconde pour être traitées, j'ai modifié la norme qui voulait que 95% des transactions soient traitées en moins d'une seconde en disant, comme cela est fait dans d'autres cas ([2],[3])que 90% des transactions devaient être traitées en moins de deux secondes.

7. Implémentation des tests ET1 sur le Vax: Choix et restrictions.

7.0. Introduction.

Nous allons décrire pour chacun des tests la manière dont il a été implémenté sur le Vax.

Le langage que nous avons employé était le Cobol, qui est celui préconisé par le standard.

Nous avons été fort limité par la place disque. En effet, le plus dont nous avons pu disposer était 50.000 blocs, et ce pour un week-end. Nous avons donc dû redimensionner chacune des applications pour les adapter aux exigences d'espace-disque.

Nous avons utilisé des fichiers de deux types: séquentiels et indexés. Les fichiers séquentiels sont stockés de telle manière que l'article qui en suit physiquement un autre est aussi celui qui le suit logiquement. Les articles sont stockés dans des blocs physiques dont la taille est 512 b.

7.1. Le test Sort.

L'opération de tri a été menée sur des fichiers de tailles différentes. L'espace-disque disponible étant limité, nous n'avons pû trier un fichier de taille supérieure à 50.000 articles. Les autres avaient une taille de 10.000, 1.000 et 100 articles.

Chaque article avait une longueur de 100 bytes, dont les dix premiers constituent la clé de tri, comme prévue dans le standard [1].

Les articles sont rangés dans des blocs physiques de taille 512 b. Le chargement des fichiers se fait au moyen d'un programme Cobol (BDSORT) dont le texte est en annexe. La clé est générée à l'aide d'un générateur de nombres aléatoires, qui a été testé par ailleurs dans [11].

L'opération de tri proprement dite est réalisée par un programme cobol qui fait un appel à l'utilitaire de tri du système.

Ici, exceptée la taille des fichiers, on voit qu'il n'y a pas eu de grands choix ou restrictions à faire quant à l'impléméntation.

7.2. Le test Scan.

L'opération de Scan a été menée sur les mêmes fichiers que ceux qui ont été triés par le test Sort. Nous avions donc des tailles de 50.000, 10.000, 1.000 et 100 articles.

La structure du fichier était la suivante: chaque article avait une longueur de 100 bytes, dont les dix premiers constituent un champ numérique mis-à-jour par l'opération de Scan, comme décrit dans [1].

Les articles sont rangés dans des blocs physiques dont la taille est 512 b.

Le chargement du fichier est réalisé par le même programme que pour le test Sort: BDSORT dont le texte se trouve en annexe.

Conformément au standard, le scan consiste à lire et à mettre à jour séquentiellement un fichier séquentiel. Cette mise-à-jour globale est éclatée en plus petites 'mini-transactions' dont la taille est respectivement, 1.000, 100, 100 et 10 articles.

On voit que pour ce test-ci, il n'y a pas eu non plus de choix ou de restrictions majeures à faire par rapport au standard.

7.3. La transaction bancaire.

Pour ce benchmark-ci, il y a eu des choix à poser, à différents niveaux.

Comme le prévoit le standard, et pour exécuter ce test dans des circonstances comparables à celles vécues chez Trasys, l'application a été écrite en Cobol.

La base de données a été organisée en quatre fichiers distincts, dont voici les tailles telles qu'implémentées:

Fichier	Nombre d'articles	Taille du fichier	Taille des articles (Bytes)	Taille de la clé (Bytes)
Agence	500	0.05 Mb	100	3
Guichet	5000	0.5 Mb	100	4
Compte	500000	50 Mb	100	10
Journal	100000	5 Mb	50	--

Tab. VII - 1: Spécification des fichiers sur Vax

On voit que par rapport aux spécifications standards, la taille des fichiers a été modifié. Ceci est dû au fait que je ne disposais que de 100 Mb pour stocker mes fichiers, ce qui m'a obligé à les redimensionner. En fait, leur taille a été réduite de moitié par rapport aux tailles implémentées sur Tandem.

Comme dit plus haut, il n'y a pas sur le Vax de générateur de transactions ou de simulateur de terminaux. Je ne pouvais pas, comme chez Trasys, enregistrer les transactions à partir d'un terminal, d'en modifier le temps de réflexion avant de les faire rejouer par un nombre défini de terminaux simulés. Pour pallier à ce manque, j'ai construit mon application de telle manière qu'au lieu de lire les données de la transaction (numéro d'agence, numéro de guichet et numéro de compte) au terminal, elle lise ces données dans un fichier. Il est clair qu'en faisant cela, je m'éloigne du standard, mais c'était probablement le

seul moyen de réussir à faire tourner l'application. Comme sur le Vax, l'architecture Requester-Server n'est pas mise en oeuvre comme sur le Tandem, l'application est réalisée par un programme unique, qui suit l'algorithme suivant:

```
Init.

Loop until end.
        Read Transaction-data from file.
        Update compte.
        Update guichet.
        Update agence.
        Write journal.
        Write message to terminal.
        Wait for delay.
End-Loop.
```

Je fais alors tourner l'application sur différents terminaux. Grâce au time-stamp associé au début et à la fin de chaque transaction, je peux déterminer le throughput (nombre de transactions traitées par seconde), ainsi que le temps de réponse du système. J'ai pu faire varier le throughput en faisant tourner l'application sur plus ou moins de terminaux, et en introduisant un délai entre les transactions.

Cette manière de procéder est moins précise et plus empirique que celle mise en oeuvre sur le Tandem, mais les moyens disponibles dans l'un et l'autre environnement permettent à mon avis de justifier cette différence.

Comme sur le Tandem, j'ai remarqué que la plupart des transactions s'exécutaient en plus ou moins une seconde pour un faible throughput; j'ai donc modifié la norme comme cela s'était déjà fait dans d'autres cas ([2], [3]), en disant que 90 % des transactions devaient être traitées en moins de deux secondes.

Voyons comment sont organisés les fichiers:

Le fichier agence est organisé de manière indexée. Il compte 500 articles. Ceux-ci sont rangés dans des buckets, comprenant 2 blocks, soit 1024 bytes (1K). Il y a un seul niveau d'index, composé à partir de la clé formée par le 3 premiers bytes de l'article. Le taux de remplissage des buckets de données est de 95%, compte tenu du fait que les données sont comprimées à 90%.

Le fichier guichet, est organisé de la même manière que le fichier agence, à ceci près que la clé est formée des 4 premiers bytes de l'article, et qu'il y a 5000 articles. Il n'y a qu'un niveau d'index, le taux de remplissage des buckets est de 99%, et les données sont comprimées à 90%.

Le fichier compte quant à lui, est composé de 500000 articles de 100 bytes, dont les 10 premiers forment la clé primaire. Ils sont stockés comme les autres dans des buckets de 1 K, à un taux de remplissage de 99%. Les records sont comprimés à 90%. Il y a trois niveaux d'index.

Notons que le taux de remplissage est très élevé. Mais comme dans notre cas, on n'ajoute pas de nouveaux records, on ne fait que des mises à jour, le risque de provoquer des bucket splits n'est pas à considérer, et l'on peut donc se permettre de remplir les buckets au taux le plus haut possible.

Faisons remarquer aussi que les clés sont attribuées aux articles par compostage; les agences sont numérotées de 0 à 499, les guichets de 0 à 4999, et les comptes de 0 à 499999. Cependant, comme les articles sont accédés dans un ordre aléatoire, le fait que les clés soient ainsi calculées n'est pas de grand intérêt.

Le fichier journal, quant à lui, compte 100000 articles, d'une longueur de 50 bytes. Il s'agit d'un fichier séquentiel, qui nécessite 9766 blocks de 512 bytes. Ici aussi, le taux de remplissage est très élevé.

Il est interressant de voir qu'il y a des 'points chauds' dans ces fichiers: d'abord, la fin du fichier journal où l'on vient écrire à la fin de chaque transaction. Ensuite, le fichier agence, qui étant de taille réduite, peut tenir en mémoire centrale. Il peut être important de trouver une solution tenant compte de ces 'points chauds' afin d'optimiser les performances de l'application [8].

8. Présentation et exploitation des résultats obtenus.

8.0. Introduction.

Nous allons maintenant examiner les résultats que nous avons obtenus lors de l'exécution des tests sur les deux machines. Cet examen se fera test par test, et nous essaierons de rechercher les différents facteurs permettant d'expliquer les différences observées entre les résultats fournis par les deux ordinateurs. Nous tenons toutefois à signaler que nous n'avons pas la prétention de dresser une liste exhaustive des facteurs pouvant expliquer les différences de performances observées entre les machines. Nous essaierons plutôt de nous attacher aux facteurs qui nous paraissent les plus déterminants. Signalons aussi que nous avons mené ce travail sur base d'une documentation très réduite; ceci signifie que nos interprétations n'ont pas toujours pu trouver un appui solide et restent par ce fait même au stade d'hypothèses.

En parallèle avec cette interprétation des résultats, nous essaierons de voir s'il n'est pas possible, et si c'est le cas, de quelle manière, d'améliorer les performances. Cette démarche se distingue de celle présentée précédemment car elle suit maintenant les phases d'implémentation, de test, de prise de mesures et de recherche de facteurs déterminants dans les performances des systèmes. Ces phases ont permis de mettre en lumière certains aspects critiques propres à chacun des systèmes examinés, et nous prendrons ces aspects comme base pour chercher des moyens d'améliorer les performances des systèmes, là où cela est possible.

8.1. Le test SORT.

8.1.1. Présentation des résultats.

D'après le standard, la mesure à prendre lors de ce test était le temps écoulé lors du tri d'un fichier. Voici le tableau présentant le temps de tri total

et par article, en fonction de la taille du fichier et de l'ordinateur observé. Le temps de tri total pour les deux ordinateurs a été mis en graphique, et les deux courbes ont été ramenées sur un seul graphe. Le résultat pour le fichier d'un million d'articles n'apparaît pas sur le graphique, car le graphe perdait sa précision pour les premières résultats. Une échelle logarithmique aurait pu être utilisée, mais nous pensons que la lecture du graphe serait devenue malaisée.

Taille du fichier (Nb articles)	Temps de tri total sur Tandem	Temps de tri par article sur Tandem	Temps de tri total sur Vax	Temps de tri par article sur Vax
100	6.1 s	61 ms	1.4 s	14 ms
1000	8.4 s	8.4 ms	3.4 s	3.4 ms
10000	38.7 s	3.87 ms	25 s	2.5 ms
50000			141 s	2.82 ms
100000	240 s	2.4 ms		
1000000	2412 s	2.412 ms		

Tab. VIII - 1: Résultats des mesures pour le test Sort.

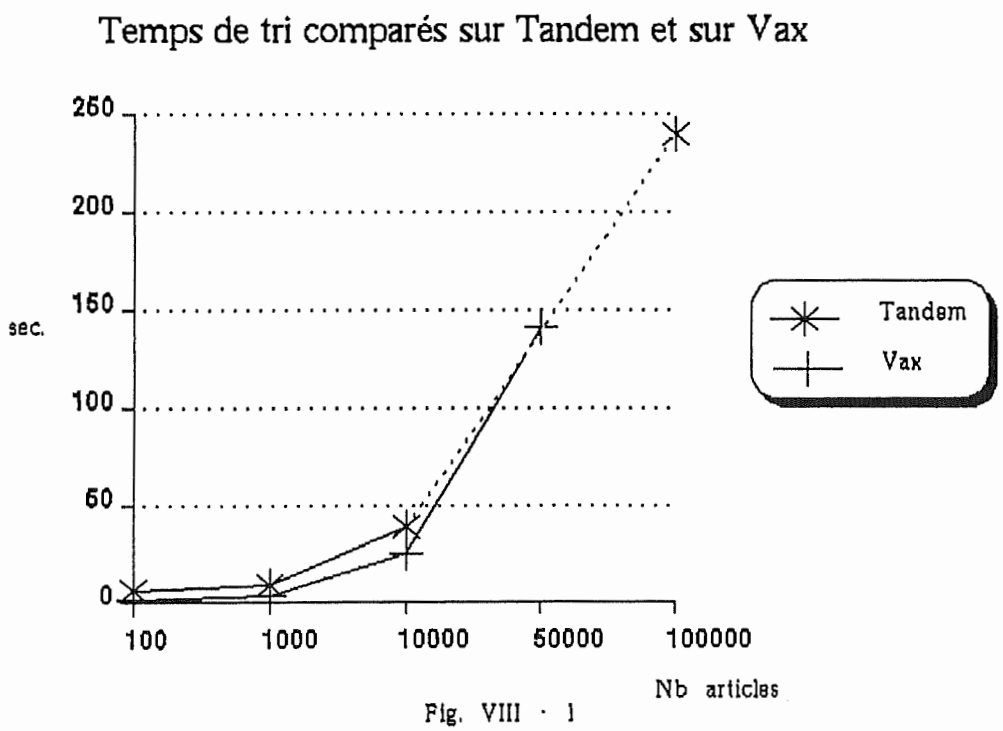


Fig. VIII - 1

On voit que pour des tailles inférieures à 10000 articles (fichiers très petits), le Vax offre de meilleures performances que le Tandem. Pour un fichier de 50000 articles, le Vax semble rejoindre le Tandem, avec la volonté de le dépasser rapidement dès que les fichiers deviennent plus volumineux.

Il est aussi interessant de voir le temps de tri ramené à un article (=temps de tri total / Nb. d'articles). C'est ce qui représenté par les deux graphiques qui suivent, l'un pour le Tandem et l'autre pour le Vax.

Temps de tri par article en fonction de la taille du fichier sur Tandem

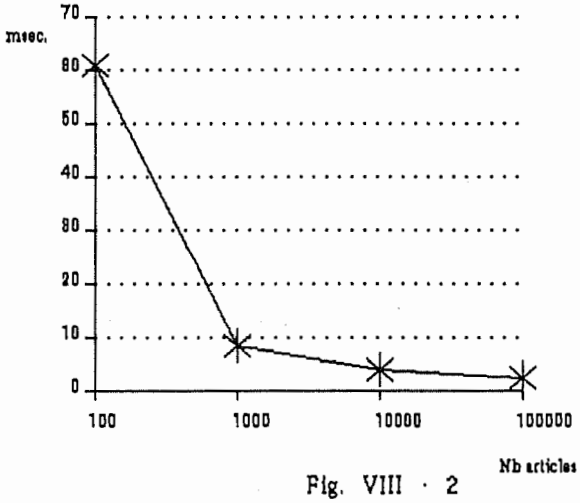


Fig. VIII · 2

Temps de tri par article en fonction de la taille du fichier sur Vax

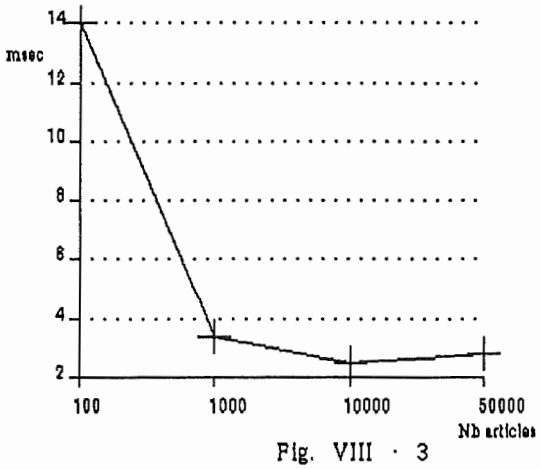


Fig. VIII · 3

Comme nous l'avons dit précédemment, nous n'avons pu mener ce test pour des fichiers de taille semblable, en raison de l'espace-disque fort limité dont nous disposions sur le Vax. Il en résulte que, pour une taille de fichier supérieure à 50.000 articles, nous n'avons pas de base de comparaison. Nous allons essayer d'extrapoler les performances du Vax pour des fichiers de taille supérieure, sur base des performances mesurées pour des plus petits fichiers, de la littérature quant au processus de tri sur Vax et quant à l'estimation théorique des performances d'un processus de tri, et d'une analyse théorique du processus de tri.

Les constructeurs affirment que le temps nécessaire au tri d'un fichier, croît de manière linéaire avec la taille de ce fichier, pour autant que la taille du fichier à trier soit suffisamment importante. Le résultat des mesures prises ne nous permet pas de contredire cette hypothèse de linéarité.

Les deux ordinateurs utilisent le même procédé de tri; ils chargent en mémoire centrale un certain nombre d'articles, les trient à l'aide d'un arbre et créent ainsi des monotonies. Chaque monotonie est sauvée sur disque, dans un fichier distinct. Une fois que tout le fichier de départ a été lu et divisé en monotonies, celles-ci sont alors fusionnées.

Le graphique suivant schématise le procédé:

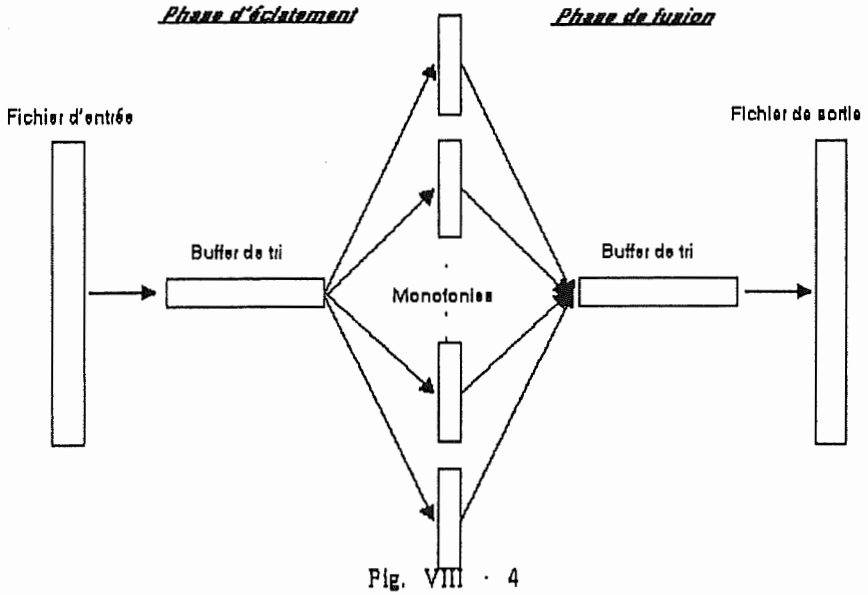


Fig. VIII - 4

8.1.2. Analyse théorique.

Soit un fichier contenant N articles.

Soit un buffer de lecture/écriture pouvant contenir N_{blo} articles.

Soit un buffer de tri pouvant contenir N_{bs} articles.

Knuth a montré que la taille des monotonies est en moyenne égale à deux fois la taille du buffer de tri. Une monotonie contiendra donc en moyenne N_m articles où $N_m=N_{bs}*2$. Le processus de tri créera donc en moyenne M monotonies, où M vaut le plus petit entier supérieur ou égal au rapport (N/N_m).

Le processus de tri travaille de manière telle que l'éclatement du fichier d'entrée se fait en une seule phase et que les monotonies créées sont sauvées dans des fichiers distincts; il y aura M fichiers créés.

Le coût en temps de la phase d'éclatement est:

$C_{pe} = C_l + C_e + C_{cpu}$

où C_l est le temps de lecture des articles du fichier
C_e est le temps d'écriture des articles dans les monotonies
C_{cpu} est le temps de traitement CPU nécessité par le tri des articles en mémoire centrale.

Le coût en temps de la phase de fusion est:

$C_{pf} = C_l + C_e + C_{cpu}$

où C_l est le temps de lecture des articles dans les monotonies
C_e est le temps d'écriture des articles dans le
C_{cpu} est le temps de traitement CPU nécessité par le tri des articles venant des monotonies en mémoire centrale.

Le coût total de la procédure de tri est donc

$$C_t = C_{ps} + C_{pf}$$
$$= 2 * (C_l + C_e + C_{cpu})$$
$$= 2 * N + (C_{ll} + C_{el}) + K_1$$

où K_1=le temps CPU nécessité par les deux phases.
C_{ll} et C_{el} sont les coûts de lecture et d'écriture d'un article.

$$= N * K_2 + K_1$$ où $K_2=2*(C_{ll} + C_{el})$

Une première remarque à faire est de dire que l'algorithme de fusion a une limite quant au nombre de monotonies qu'il peut fusionner en même temps. Si la taille du fichier à trier est telle que le nombre de monotonies créées dépasse cette limite, la fusion se fera en au moins deux passes. Cette limite peut être imposée soit par l'algorithme de fusion lui-même, de par sa conception, soit par le système, qui impose un nombre maximum de fichiers ouverts en même temps. L'algorithme de fusion du Vax a un ordre maximum égal à 10. Si le nombre de monotonies à fusionner est supérieur à cette limite, il faudra donc fusionner en au moins deux passes. Vu la taille des fichiers que nous avons eu à trier, une seule passe suffisait, aussi bien lors de la phase d'éclatement que lors de la phase de fusion, ce qui signifie que nous n'avons pu apprécier la différence de performance lorsque plusieurs passes sont nécessaires.

Détaillons un peu chacun des coûts envisagés:

a) les coûts de lecture des articles.

Selon la taille du fichier à trier, une seule lecture des articles suffira, car tous pourront être chargés en mémoire centrale, ou plusieurs lectures seront nécessaires, car il faudra créer des monotonies, et donc relire celles-ci. Cependant, la lecture s'effectue dans un buffer de lecture, pouvant contenir

N_{bio} articles. Le nombre d'accès physiques en lecture aux disques, nécessaires à la lecture de tous les articles du fichier, sera égal au rapport (N/N_{bio}). Soit N_{apl}, ce résultat. Si la taille du fichier est telle que tous les articles peuvent être triés en mémoire centrale, une seule phase de lecture sera nécessaire, et le coût total de lecture sera:

$$C_l = N_{apl} * \left(\text{Temps d'accès} + \frac{N_{bio} * \text{Taille d'un article}}{\text{Vitesse de transfert du disque}} \right)$$

Formule $F_{8.1}$

Si la taille du fichier est telle que tout le fichier ne peut être directement placé en mémoire centrale, le coût total en lecture sera:

$$C_l = N_{apl} * \left(\text{Temps d'accès} + \frac{N_{bio} * \text{Taille d'un article}}{\text{Vitesse de transfert·disque}} \right) + M * \frac{N_m}{N_{bio}} * \left(\text{Temps d'accès} + \frac{N_{bio} * \text{Taille d'un article}}{\text{Vitesse de transfert·disque}} \right)$$

Formule $F_{8.2}$

Nous ne considérons ici que le cas où une seule passe est nécessaire pour fusionner les monotonies créées. Si ce n'est pas le cas, le résultat peut être aisément adapté.

b) Les coûts d'écriture des articles.

Le calcul des coûts d'écriture des articles peut être réalisé en suivant le même raisonnement que pour la lecture des articles. En effet, les écritures se font via un buffer.

Si le fichier contient N articles, le nombre d'accès disque nécessaires pour l'écriture de tous les articles sera égal au rapport (N/Nbio). Soit N_{ape}, ce résultat. Si la taille du fichier est telle que tout le fichier tient en mémoire, une seule écriture de tous les articles sera nécessire. Le coût d'écriture sera égal au rapport suivant, sous l'hypothèse qu'un accès en écriture prend le même temps qu'un accès en lecture.

$$C_e = N_{ape} * \left(\text{Temps d'accès} + \frac{N_{bio} * \text{Taille d'un article}}{\text{Vitesse de transfert du disque}} \right)$$

Formule $F_{(8.3}$

Si la taille du fichier est telle qu'il ne peut tenir en entier en mémoire centrale, et que la création de monotonies est nécessaire, le coût total en écriture sera égal à l'expression suivante, sous la même hypothèse que celle exprimée ci-dessus.

$$C_e = N_{ape} * \left(\text{Temps d'accès} + \frac{N_{bio} * \text{Taille d'un article}}{\text{Vitesse de transfert-disque}} \right)$$

$$+ M * \frac{N_m}{N_{bio}} * \left(\text{Temps d'accès} + \frac{N_{bio} * \text{Taille d'un article}}{\text{Vitesse de transfert-disque}} \right)$$

Formule $F_{(8.4}$

Encore une fois, nous ne considérons pas ici le cas où plusieurs passes sont nécessaires, lors de la phase de fusion. Toutefois, le résultat ci-dessus pourra être facilement adapté à ce cas.

c) Discussion des coûts de lecture/écriture.

Nous avons vu que le nombre de monotonies M, croît de manière linéaire avec la taille du fichier. Il en va donc de même pour le nombre d'accès physiques en lecture et en écriture, et donc aussi pour le coût total en lecture ou en écriture. Rappelons que ces affirmations ne sont vérifiées que dans le cas où une seule passe suffit lors de la phase de fusion. On peut donc dire que les coûts de lecture et d'écriture varient de manière linéaire avec la taille. Ce résultat rejoint l'étude réalisée dans l'article [6].

d) Les coûts de tri des articles en mémoire centrale.

Malgré le conclusion de linéarité des coûts de lecture et d'écriture sur disque, il intervient dans le calcul du coût total du tri une composante, qui ne varie pas de manière linéaire; il s'agit du processus de tri interne, qui utilise un arbre binaire pour arranger les articles en mémoire centrale. Ce processus a en effet une complexité de N*Log(N), où N est la taille de l'arbre. Tâchons d'expliquer un peu pourquoi. Les articles sont rangés dans un arbre binaire de la manière suivante:

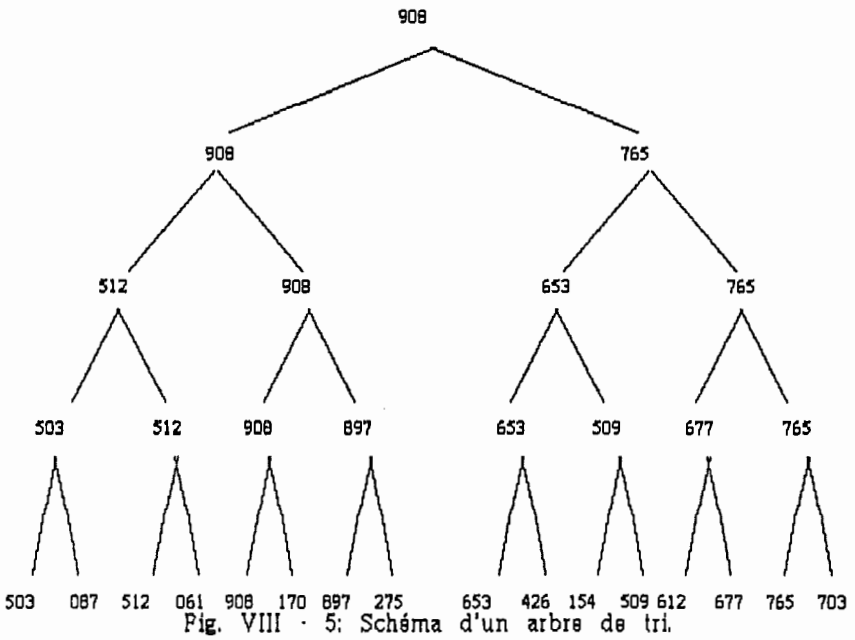


Fig. VIII - 5: Schéma d'un arbre de tri.

Une fois que l'arbre est rempli, s'il reste des articles à trier, le processus lit l'article suivant dans le fichier; nous nous plaçons dans la perspective d'un tri par ordre croissant. Si le nouvel article est plus petit que la racine de l'arbre, il casse la monotonie créée. Celle-ci est donc écrite sur disque. Mais si l'article est plus grand, la racine est écrite sur disque, ce qui résulte en un trou dans l'arbre. Pour recalculer l'arbre, $\log_2(N)$ comparaisons sont nécessaires, si N est le nombre de records contenus dans l'arbre. La complexité totale de ce procédé est donc N*Log(N). Ce résultat a été expliqué par Knuth [13].

Cependant, tant que la taille de l'arbre de tri n'excède pas 10000 articles, la composante N Log(N) ne domine pas les coûts de lecture/écriture des articles [6].

8.1.3. Le cas particulier du Tandem.

Le Tandem fait usage de différentes 'astuces' pour améliorer les performances de son processus de tri. Il réalise les entrées/sorties via un buffer de 6 Kb, il alloue la mémoire en fonction de la taille du fichier à trier, pour permettre de réaliser le tri en une seule passe, aussi bien lors de la phase d'éclatement que lors de la phase de fusion. Comme spécifié plus haut, les disques du Vax offrent une vitesse d'accès (délai rotationnel + seek time) de 24.8 ms et une vitesse de transfert de 1.860Mb/s. Une autre caractéristique du Tandem est d'utiliser différents processeurs en parallèle pour réduire le temps de traitement. Nous n'avons pu exploiter cette caractéristique, car nous ne pouvions disposer que d'un seul processeur. La mémoire allouée au processus de tri permettait de trier en mémoire centrale des fichiers contenant jusqu'à 10000 articles. Nous ne considèrerons pas le cas où plusieurs passes sont nécessaires au processus de fusion pour créer le fichier de sortie. De plus, nous prendrons l'hypothèse que le coût d'une lecture sur disque est le même que le coût d'une écriture.

Nous allons maintenant détailler chacun des coûts développés ci-dessus en les instanciant au cas particulier du Tandem.

a) Les coûts de lecture des articles.

Les articles sont rangés dans des blocs de 1024 B, avec une moyenne de 9 articles par bloc. Le buffer d'entrée/sortie a une taille de 6Kb, soit 6 blocs ou encore 54 articles. Si le fichier peut tenir entièrement en mémoire centrale, le temps total de lecture du fichier peut être évalué par l'expression suivante qui utilise la formule F_{81}. Nous prenons pour l'exemple le cas d'un fichier de 1000 articles.

$$C_1 = \frac{1000}{54} * (24.8 \text{ ms} + \frac{54*100B}{1860B/ms})$$

$$= 18.5 * (24.8 \text{ ms} + 2.903 \text{ ms})$$

$$= 512.5 \text{ ms}$$

Si par contre, la taille du fichier ne permet pas de le trier entièrement en mémoire centrale, il faudra donc créer des monotonies, et par conséquent, il faudra les relire. Dans ce cas, une bonne estimation du temps passé en lecture sera donnée par l'expression suivante, qui utilise la formule F_{82}. Nous prenons pour l'exemple, le cas d'un fichier de 100000 articles.

$$C_1 = \frac{100000}{54}*(24.8ms + \frac{54*100B}{1860B/ms}) + 10* \frac{100000}{54} *(24.8ms + \frac{54*100B}{1860B/ms})$$

$$= 102500 \text{ ms}$$

Si l'on refait le même calcul pour les autres fichiers, on obtient le tableau ci-dessous:

Taille du fichier (Nb articles)	Estimation du temps de lecture
100	51.3 ms
1000	512.5 ms
10000	5129.6 ms
50000	51302 ms
100000	102500 ms
1000000	1025925 ms

Tab. VIII - 2: Estimation du temps de lecture.

b) Les coûts d'écriture des articles.

La taille du buffer d'entrée/sortie est comme nous l'avons dit de 6 Kb. Lors d'un processus de tri, les écritures ne sont pas dédoublées, car cela prendrait trop de temps. Si le fichier à trier tient entièrement en mémoire centrale, il n'y a pas besoin de créer des monotonies. Les seules écritures seront celles des articles dans le fichier de sortie. Les coûts d'écriture peuvent être estimés grâce à la formule F_{83}. C'est ce qui est fait dans l'expression ci-dessous, pour le cas d'un fichier de 1000 articles:

$$C_8 = \frac{1000}{54} * \left(24.8 \text{ ms} + \frac{54 * 100B}{1860B/ms}\right)$$

$$= 18.5 * (24.8 \text{ ms} + 2.903 \text{ ms})$$

$$= 512.5 \text{ ms}$$

Si maintenant, la taille du fichier ne permet pas de trier celui-ci entièrement en mémoire, il faudra générer des monotonies, qui devront être écrites sur disque. Il faut donc rajouter au temps d'écriture du fichier de sortie, le temps d'écriture des monotonies. C'est ce qui est réalisé par l'expression ci-dessous, en se basant sur la formule F_{84}. Nous avons considéré le cas d'un fichier de 100000 articles.

$$C_8 = \frac{100000}{54} * (24.8ms + \frac{54*100B}{1860B/ms}) + 10 * \frac{10000}{54} * (24.8ms + \frac{54*100B}{1860B/ms})$$

$$= 102500 \text{ ms}$$

Le tableau qui suit présente une estimation du temps passé en écriture pour chacun des fichiers, ainsi que du temps total passé à effectuer des opérations d'entrée/sortie.

Taille du fichier (Nb articles)	Estimation du temps d'écriture	Estimation du temps total I/O
100	51.3 ms	102.6 ms
1000	512.5 ms	1025 ms
10000	5129.6 ms	10259.2 ms
50000	51302 ms	102604 ms
100000	102500 ms	205000 ms
1000000	1025925 ms	2051850 ms

Tab. VIII - 3: Estimation du temps total I/O

c) Le coût de tri des articles en mémoire centrale.

Pour cet aspect du coût total, nous ne disposons pas de mesures précises permettant de déterminer le temps CPU effectivement consommé par le processus de tri.

d) Réflexion sur le résultat des estimations.

On voit que les estimations réalisées se rapprochent assez fort des mesures prises. Cependant, rappelons qu'elles ne tiennent compte que du temps passé en entrées/sorties. Il faut rajouter à ce temps le temps CPU nécessité par le tri des articles en mémoire. Toutefois, les estimations calculées ci-dessus restent du même ordre que les mesures prises lors des tests. Les différences constatées peuvent s'expliquer par l'overhead causé parl'ouverture et la fermeture des différents fichiers utilisés, auquel le temps d'exécution d'un processus sur un petit fichier est probablement plus sensible, qu'un processus travaillant sur un fichier de taille plus importante. Il faut aussi signaler que ce que nous avons calculé ne sont que des estimations des temps d'exécution, et n'ont donc de ce fait qu'une valeur indicative. Cependant, il semble qu'elles fournissent une assez bonne appréciation de la réalité.

8.1.4. Le cas particulier du Vax.

Le Vax utilise son architecture software I/O pour minimiser les temps de tri. La lecture d'un fichier se fait par 16 blocs en une fois, la mémoire allouée au processus de tri est calculée au mieux des possibilités, pour éviter des échanges inutiles avec le disque. La mémoire allouée permet de trier en mémoire centrale des fichiers jusqu'à 34000 articles. Au-delà de ce nombre, il faut créer des monotonies. Les disques du Vax offrent une vitesse d'accès (délai rotationnel + seek time) de 36.3 ms et une vitesse de transfert de 2.20 Mb/s. Nous allons reprendre ici chacun des coûts évoqués plus haut, en les instanciant au cas particulier du Vax. Signalons cependant que les résultats calculés constituent des estimations des valeurs réelles, car nous nous sommes basés pour les calculs sur des valeurs moyennes, pour le nombre d'articles par bloc, pour la taille des monotonies, ...

a) Les coûts de lecture des articles.

Nous savons que les articles sont rangés dans des blocs de 512 B, et que ces blocs sont lus par ensemble de 16. Si la taille du fichier est telle qu'il peut être trié entièrement en mémoire centrale, le temps total de lecture du fichier sera égal à l'expression suivante pour le cas d'un fichier contenant 1000 articles, en utilisant la formule F_{81}.

$$C_l = \frac{1000}{80} * (36.3\ ms + \frac{80 * 100B}{2200B/ms})$$

$$= 12.5 * (36.3\ ms + 3.63\ ms)$$

$$= 499.1\ ms$$

Si la taille du fichier ne permet pas de le trier entièrement en mémoire centrale, il faudra donc créer des monotonies. Nous prendrons l'hypothèse que le nombre de monotonies créées n'excède pas l'ordre de l'algorithme de fusion, ce qui signifie que le fichier sera lu une première fois entièrement, et que chaque monotonie sera lue une et une seule fois. Si nous considérons le cas d'un fichier contenant 50000 articles, le processus de tri créera deux monotonies (ce résultat est tiré de l'analyse des mesures prises lors de l'exécution du test). Le temps passé en lecture sera alors égal à l'expression suivante, en utilisant la formule F_{82}:

$$C_l = \frac{50000}{80} * (36.3ms + \frac{80*100B}{2200B/ms}) + 2* \frac{25000}{80} * (36.3ms + \frac{80*100B}{2200B/ms})$$

$$= 49920\ ms$$

Si l'on refait le même calcul pour les autres fichiers, on obtient le tableau suivant, sachant que les fichiers contenant jusqu'à 10000 articles pouvaient être triés directement en mémoire centrale.

Taille du fichier (Nb articles)	Estimation du temps de lecture
100	49.9 ms
1000	499.1 ms
10000	4991.2 ms
50000	49920 ms

Tab. VIII · 4: Estimation du temps de lecture

b) Les coûts d'écriture des articles.

L'écriture des articles se fait aussi via un buffer pouvant contenir 16 blocs. Dès lors, si la taille du fichier permet de le trier entièrement en mémoire, il en résulte qu'une seule écriture de tous les articles est nécessaire. Nous prenons l'hypothèse que le coût d'une écriture est égal au coût d'une lecture. Le temps nécessité par l'écriture d'un fichier trié de 1000 articles sera égal à l'expression suivante, en utilisant la formule F_{83}:

$$C_8 = \frac{1000}{80} * \left(36.3\ ms + \frac{80 * 100B}{2200B/ms}\right)$$

$$= 12.5 * (36.3\ ms + 3.72\ ms)$$

$$= 499.1\ ms$$

Si la taille du fichier justifie la création de monotonies, il faut encore écrire ces monotonies sur disque. Il faut donc rajouter à l'écriture du fichier de sortie, l'écriture de chacune des monotonies. C'est ce qui est fait dans

l'expression suivante, pour le cas d'un fichier de 50000 articles, en utilisant la formule F_{84}:

$$C_8 = \frac{50000}{80} * \left(36.3ms + \frac{80*100B}{2200B/ms}\right) + 2* \frac{25000}{80} * \left(36.3ms + \frac{80*100B}{2200B/ms}\right)$$

= 49920 ms

Si l'on refait le même calcul pour les autres fichiers, on trouve le tableau suivant, qui calcule en même temps le temps total passé par le processus de tri en I/O:

Taille du fichier (Nb articles)	Estimation du temps d'écriture	Estimation du temps total I/O
100	49.9 ms	99.8 ms
1000	499.1 ms	998.2 ms
10000	4991.2 ms	9982.4 ms
50000	49920 ms	99840 ms

Tab. VIII - 5: Estimation du temps total I/O

c) Le coût de tri des articles en mémoire centrale.

Pour cette composante du coût total, nous avons obtenu pour chacun des fichiers triés, le temps CPU exact utilisé par le processus. C'est ce résultat que présente le tableau suivant, en calculant par la même occasion, l'estimation du temps total nécessaire à l'exécution du processus de tri.

Taille du fichier (Nb articles)	Estimation du temps CPU utilisé	Estimation du temps total I/O	Estimation du temps d'exécution
100	520 ms	99.8 ms	619.8 ms
1000	1860 ms	998.2 ms	2828.2 ms
10000	13200 ms	9982.4 ms	23182.4 ms
50000	59440 ms	99840 ms	159280 ms

Tab. VIII - 6: Estimation du temps total d'exécution.

d) Réflexion sur le résultat des estimations.

On voit que les estimations du temps nécessaire au processus de tri sont assez proches des mesures prises lors de l'exécution du test. On constate cependant certaines divergences entre les mesures prises lors de l'exécution des tests et les estimations calculées ci-dessus. Ces différences peuvent s'expliquer par différents facteurs, comme expliqué plus haut lors de la réflexion sur les résultats des estimations calculées sur Tandem; il s'agit donc de l'overhead d'ouverture/fermeture des fichiers, et du fait que ce que nous avons calculé ne sont que des estimations, et n'ont de ce fait qu'une valeur indicative.

8.1.5. Conclusion.

En guise de conclusion de l'interprètation des résultats de ce test, nous pouvons dire que les deux machines se ressemblent fortement du point de vue des performances. L'aspect du système que le test voulait éprouver était l'architecture hardware I/O. Il semblerait que les deux architectures soient plus ou moins de même force, dans le cadre du tri de fichier. Ceci s'explique probablement par le fait que les deux systèmes utilisent le même procédé de tri, et quand l'un charge des ensembles plus grands de blocs, ceux-ci sont plus petits que sur l'autre système. Et si l'un a une vitesse de transfert du disque vers la mémoire supérieure, l'autre compense par une vitesse d'accès plus élevée. Les forces s'équilibrent, ce qui résulte en des performances plus ou moins semblables.

8.2. Le test SCAN.

8.2.1. Présentation des résultats.

D'après le standard, la mesure à prendre pour ce test était aussi le temps écoulé. Le tableau qui suit reprend le temps total nécessité pour la mise-à-jour du fichier global, et le temps nécessité pour la mise-à-jour d'une mini-transaction en fonction de la taille des fichiers et de l'ordinateur observé. Pour rappel, une mini-transaction consiste en la mise-à-jour d'un sous-ensemble d'un certain nombre d'articles (10,100 ou 1000 selon les cas).

Taille du fichier (Nb articles)	Temps de Scan total sur Tandem	Temps de Scan total sur Vax
100	4.5 s	1.12 s
1000	30.4 s	3.17 s
10000	294 s	15.69 s
50000		69.6 s
100000	2898 s	
1000000	2960 s	

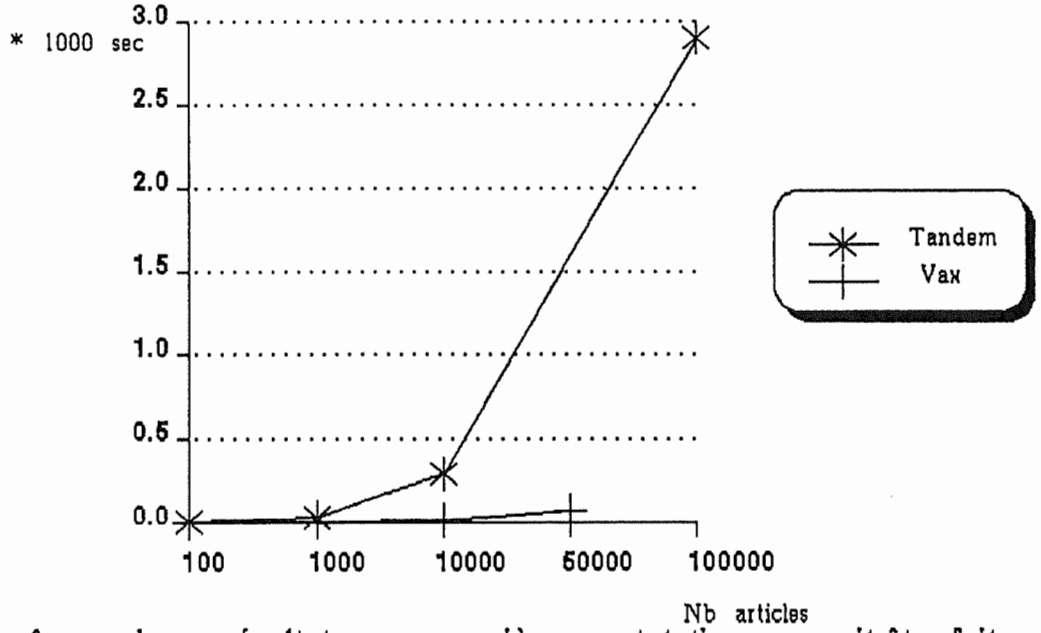
Tab. VIII - 7: Temps d'exécution du test Scan.

Taille du fichier (Nb articles)	Taille de la mini-transaction (Nb articles)	Temps de Scan d'une mini-transaction sur Tandem	Temps de Scan d'une mini-transaction sur Vax
100	10	450 ms	112 ms
1000	100	3040 ms	317 ms
10000	100	2940 ms	156 ms
50000	1000		1380 ms
100000	1000	28980 ms	
1000000	1000	29160 ms	

Tab. VIII - 8: Temps de traitement d'une mini-transaction

Le graphique qui suit établit une comparaison entre le temps de mise-à-jour de tout le fichier nécessité par le Vax et le Tandem.

Temps de Scan comparés sur Tandem et sur Vax



Au vu de ces résultats, une première constatation pourrait être faite, qui consisterait à dire que le Vax est nettemment plus rapide que le Tandem; ce test de Scan étant prévu pour évaluer les performances du software I/O du système, il semblerait alors que " l'architecture " software I/O du Vax soit nettement meilleure que celle du Tandem. Par une analyse plus approfondie, nous nous attacherons à rechercher les facteurs déterminants qui permettent d'expliquer les différences de performance entre les deux machines.

8.2.2. Facteurs communs aux deux systèmes.

Pour comprendre la différence des performances mesurées sur l'une et l'autre des machines, il est important de détailler l'algorithme suivi: le test consiste en la mise-à-jour séquentielle d'un fichier organisé de manière

séquentielle. Les articles mis à jour sont protégés par un simple blocage en lecture/écriture, le temps de l'opération. L'ordinateur va donc chercher l'article sur le disque, le met à jour, et le ré-écrit sur le disque. Cependant, les articles sont sauvés sur disque dans des blocs, de taille différente: sur le Tandem, les blocs ont une taille de 1024 bytes, alors que sur le Vax, leur taille est de 512 bytes. Comme la lecture/mise-à-jour se fait de manière séquentielle, un nouveau bloc n'a pas besoin d'être chargé chaque fois que l'on veut accéder à un nouvel article.

En effet, sur le Tandem, un bloc contient 9 articles; le nombre de blocs à charger sera donc le résultat du rapport:

$$\frac{\text{Nombre d'articles dans le fichier}}{9}$$

Sur le Vax, un bloc contient en moyenne 5 articles; en effet, le Vax permet qu'un article soit écrit à cheval sur deux blocs (ce que le Tandem ne permet pas). Le nombre de blocs à charger, pour le Vax sera donc le résultat du rapport:

$$\frac{\text{Nombre d'articles dans le fichier}}{5}$$

Le tableau qui suit donne le nombre de blocs à charger, en fonction de la taille des fichiers et de la machine observée. Il s'agit en fait du nombre de blocs contenant les données, plus quelques blocs d'overhead, contenant des informations de contrôle et de description du fichier.

Taille des fichiers (Nb articles)	Nombre de blocs par fichier sur Tandem	Nombre de blocs par fichier sur Vax
100	12	20
1000	112	195
10000	1112	1954
50000		9765
100000	11112	
1000000	111112	

Tab. VIII - 9: Taille des fichiers implémentés.

Cette différence dans le nombre de blocs à charger, qui conditionne le nombre d'accès au disque, est déjà un facteur important dans l'explication des performances des deux systèmes.

Un autre facteur important est la vitesse d'accès au disque et sa vitesse de transfert. Le Vax offre une vitesse d'accès (Seek Time + Délai rotationnel) de 36.3 ms et un débit de 2.20 MB/s. Le Tandem quant à lui offre une vitesse d'accès de 24.8 ms et un débit de 1.86 MB/s.

Le temps de transfert d'un bloc de 1024 Bytes du disque vers la mémoire, sur le Tandem nécessite:

24.8 ms + (1024 B /1860 B/ms) = 25.35 ms

Le temps de transfert d'un bloc de 512 Bytes du disque vers la mémoire, sur le Vax nécessite:

36.3 ms + (512 B /2200 B/ms) = 36.5 ms

Ces deux facteurs porteraient plutôt à croire que le Tandem offre de meilleures performances que le Vax, ce qui s'opposerait aux résultats des mesures prises, car comme on le voit sur les graphiques précédents, le Vax offre de meilleures performances que le Tandem, compte tenu du fait que nous n'avons pu créer de fichier d'une taille supérieure à 50000 articles. Ce serait ne pas tenir compte d'un troisième facteur: la manière dont les deux ordinateurs réalisent la procédure de mise-à-jour.

8.2.3. Facteurs propres au Tandem.

Nous avons examiné les listings issus des mesures prises lors de l'exécution du test sur le Tandem. On constate que, à chaque fois qu'une

écriture sur disque est rendue nécessaire par la mise·à·jour d'un article, un accès disque est réalisé. Dès lors, si pour la lecture, qui constitue la première phase de la mise·à·jour, le nombre d'accès physiques au disque est réduit au nombre de blocs composant le fichier, pour l'écriture, il y a autant d'accès disque que d'articles mis·à·jour. Au total, le nombre d'accès physiques est donc:

Nombre de blocs par fichier + Nombre d'articles par fichier

Soit, pour un fichier de 1.000.000 articles:

Nombre de blocs: 111.112

Nombre d'articles: 1.000.000

On trouve:

Nombre d'accès physiques: 1.111.112

Si l'on multiplie ce nombre par le temps moyen de transfert d'un bloc associé aux disques du Tandem, on trouve:

Temps d'activité du disque = Temps d'activité par accès * Nombre d'accès
= 25.35 ms * 1111112
= 28166 s
= 7.82 Heures

Ce temps total d'utilisation du disque est évidemment une estimation du temps réel, car nous nous sommes basés pour le calculer, sur le temps **moyen** d'accès au disque.

Si l'on suit la même procédure, on trouve comme estimation du temps total d'activité du disque, pour les fichiers de taille 100000, 10000, 1000, 100:

Taille des fichiers (Nb articles)	Nombre total d'accès	Estimation du temps total d'activité du disque
100	112	2.9 s
1000	1112	28.2 s
10000	11112	4.7 m
100000	111112	47 m
1000000	1111112	7.8 h

Tab. VIII · 10: Estimation du temps d'activité du disque

Il faut rajouter à ces estimations, le temps CPU utilisé pour l'exécution du test. Le tableau qui suit présente le temps CPU consommé pour le Scan de chacun des fichiers, ainsi qu'une estimation du temps total d'exécution de ce test.

Taille des fichiers (Nb articles)	Temps CPU consommé	Estimation du temps total d'activité du disque	Estimation du temps total d'exécution
100	0.402 s	2.9 s	3.3 s
1000	3.4 s	28.2 s	31.6 s
10000	34.19 s	4.7 m	5.2 m
100000	338.6 s	47 m	52.6 m
1000000	3401 s	7.8 h	8.6 h

Tab. VIII · 11: Estimation du temps total d'exécution

On voit que ces temps sont très proches des temps mesurés pour l'exécution du test, tout en leur restant un petit peu supérieur. Il est probable que nous ayons légèrement surestimé le temps CPU, car les statistiques dont nous disposions n'étaient pas très précises. De même, nous avons peut-être surestimé le temps d'activité du disque. Mais rappelons que ce résultat n'est qu'une estimation. Il ne fournit à ce titre qu'un valeur indicative, autour de laquelle les valeurs des mesures prises lors de l'exécution du test gravitent. En

effet, tantôt ces mesures dépasseront ces estimations, tantôt elles resteront en-deça. Il n'en reste pas moins que nos hypothèses semblent confirmées par l'adéquation mesures/estimations.

8.2.4. Facteurs propres au Vax.

La manière de procéder du Vax est différente. Quand la lecture d'un bloc est demandée sur le disque, il charge en même temps dans un buffer, les 15 blocs suivants.

La technique du Read-Ahead/Write-Behind est utilisée pour limiter le nombre d'accès physiques au disque. Deux buffers sont utilisés, et quand l'un doit être écrit sur disque ou lu depuis le disque, l'autre prend le relais. Le système ne doit donc pas attendre que l'opération I/O soit terminée, il peut continuer sa tâche normale.

Le buffer d'entrée peut contenir 16 blocs, soit 8192 bytes. Le temps d'accès nécessité par le chargement du buffer est donné par le calcul suivant:

$$C_t = \frac{\text{Taille du buffer}}{\text{Vitesse de transfert}} + \text{Vitesse d'accès}$$

$$= \frac{8192\ B}{2200\ B/ms} + 36.3\ ms$$

$$= 40.02\ ms$$

Si l'on prend le cas du fichier de 10000 articles, composé de 1954 blocs, le temps passé en lecture sera égal à l'expression suivante:

$$C_t = \frac{\text{Nombre de blocs}}{\text{Taille du buffer en blocs}} * \text{temps de transfert du buffer}$$

$$= \frac{1954}{16} * 40.02 \text{ ms}$$

$$= 4.887 \text{ s}$$

Le temps passé en écriture peut être évalué par le même calcul, si l'on adopte l'hypothèse que le coût d'une écriture est égal au coût d'une lecture:

$$C_t = \frac{1954}{16} * 40.02 \text{ ms}$$

$$= 4.887 \text{ s}$$

Le temps total passé en entrées/sorties vaut donc:

temps total = temps de lecture + temps d'écriture
= 4.887 s + 4.887 s
= 9.775 s

Taille du fichier (en blocs)	Temps passé en lecture	Temps passé en écriture	Temps total passé en I/O
20	50.02 ms	50.02 ms	100.05 ms
195	487.7 ms	487.7 ms	975.4 ms
1954	4887.4 ms	4887.4 ms	9774.8 ms
9765	24424.7 ms	24424.7 ms	48849.4 ms

Tab. VIII - 12: Temps total I/O

On peut refaire le même calcul pour les autres tailles de fichiers:

Nous avons donc calculé le temps total passé par le processus de Scan en entrées/sorties disque. A ce temps, il faut encore rajouter le temps CPU utilisé par le processus. Ce temps n'est pas négligeable, surtout quand la taille du fichier augmente. Le tableau qui suit présente les résultats du calcul du temps total d'exécution du processus:

Taille du fichier (Nb articles)	Temps total passé en I/O	Temps CPU total	Temps total
100	100.05 ms	320 ms	420 ms
1000	975.4 ms	1800 ms	2775 ms
10000	9774.1 ms	6700 ms	16474 ms
50000	48849.4 ms	19520 ms	68369 ms

Tab. VIII · 13: Temps total d'exécution

On voit que ces chiffres sont très proches des résultats des mesures prises lors de l'exécution du test. Cependant, pour les petites tailles de fichier, la différence est plus marquée. Cela peut s'expliquer à différents niveaux: d'abord, il faut tenir compte de l'overhead causé par l'ouverture et la fermeture des fichiers. Si cet overhead peut être dissous dans le temps d'exécution total lorsque celui-ci est important, quand ce n'est pas le cas, il devient beaucoup plus perceptible. Ensuite, il faut dire que si les blocs sont lus par ensemble de 16, le premier fichier ne contient que 20 blocs. La première lecture en chargera donc 16 et la seconde, 4. Notre calcul ci-dessus était donc un peu biaisé, car au lieu de 1.25 lectures comme nous l'avions calculé (20/16), il se produit bien deux lectures. Et comme le facteur prépondérant dans le temps de transfert reste la vitesse d'accès à l'information voulue, le temps de lecture de tout le fichier sera donc supérieur aux 50.025 ms que nous avions calculé ci-dessus. Ce phénomène est observé, dans une moindre mesure, dans le cas du fichier de 1000 articles.

8.2.5. Conclusion.

La conclusion de l'interprètation des résultats du test Scan, suit logiquement l'évolution de ces résultats: l'ordinateur Vax offre de meilleures performances que le Tandem, du point de vue du software I/O. Et ceci montre que le test Scan, dans sa conception, est bien destiné à tester le software I/O du système. En effet, l'aspect hardware a été contourné, comme l'on peut s'en rendre compte en constatant que l'architecture hardware proprement dite favorisait plutôt le Tandem, qui dispose de disques plus rapides, plus performants,... Mais le software du Vax permet d'améliorer les performances, même avec du hardware relativement moins performant

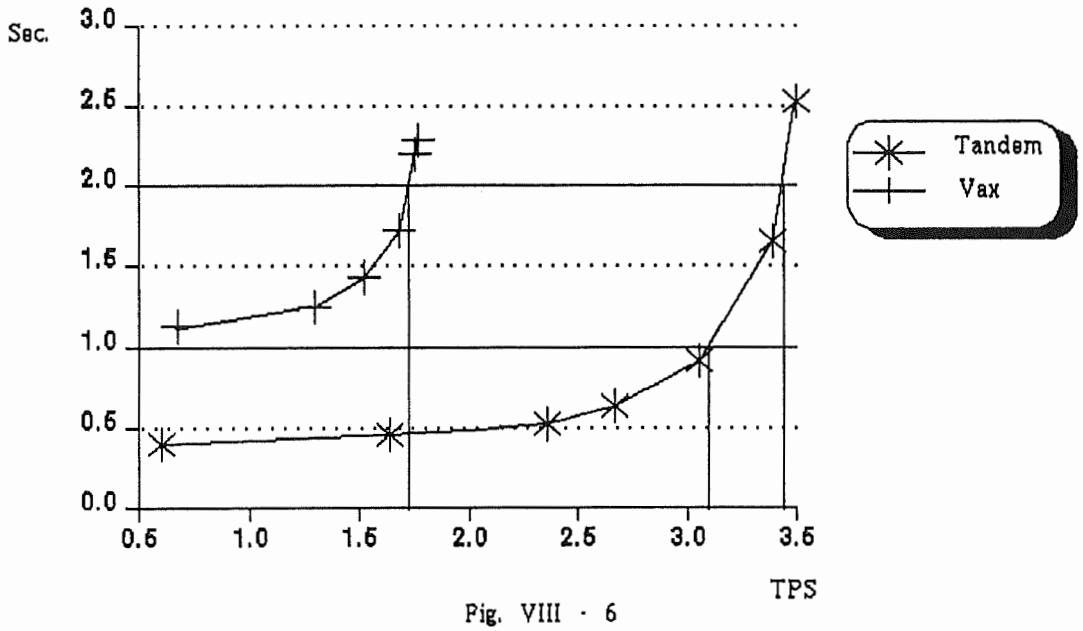
8.3. La transaction bancaire.

8.3.1. Présentation des résultats

Comme décrit plus haut, pour faire croître le nombre de transactions à traiter, j'ai dû utiliser deux méthodes différentes: sur le Tandem, j'ai utilisé l'outil de simulation de terminaux (ENCORE). Malheureusement, cet outil ne peut simuler que 12 terminaux à la fois. Ceci ne nous amenait qu'à un nombre de transactions par seconde assez faible (1.16 avec un Think Time de 10 sec.) et ne nous permettait en aucun cas de tirer des conclusions. Pour remédier à cela, nous avons fait augmenter artificiellement le nombre de transactions par seconde en faisant décroître le temps de réflexion (Think Time) sur un nombre fixe de terminaux (10). Sur le Vax, j'ai fait tourner l'application sur un nombre fixe de terminaux, en mesurant à chaque étape, le throughput et le temps de réponse. Pour voir l'évolution des performances, j'ai ensuite fait varier le nombre de terminaux. Ces deux méthodes nous ont permis de tirer des résultats, qui gardent quand même, de par la méthode employée, une part d'imprécision, à partir desquels nous avons essayer d'établir une courbe d'évolution du temps de réponse en fonction du nombre de TPS (Transaction Par Seconde).

Voici un graphique qui reprend les courbes du temps de réponse donné par le système, en fonction du nombre de transactions traitées par seconde, pour chacun des deux ordinateurs observés.

Temps de réponse comparés en fonction du throughput
Sur Tandem et sur Vax.



Pig. VIII - 6

A la vue de ce graphique, une constatation s'impose: le Vax s'essouffle très rapidement. Ainsi, on voit que quelque soit le throughput, le temps de réponse est au dessus de la seconde. Et quand le throughput atteint 1.75 TPS, le temps de réponse devient supérieur à deux secondes.

Le Tandem présente une courbe plus intéressante. En effet, tant que le throughput reste inférieur à 3.1 TPS, le temps de réponse reste inférieur à une seconde. Et quand le throughput atteint 3.4 TPS, le temps de réponse dépasse les deux secondes.

A ce stade, nous voudrions faire une remarque. La firme Tandem affirme que si l'on accroît le matériel installé dans une certaine proportion, les

performances croissent dans le même rapport. Or, nous avons fait tourner notre application sur un processeur TXP, et nous obtenons comme throughput assurant un temps de réponse inférieur à deux secondes, 3.4 TPS. Le même test a été effectué par la firme Tandem sur 32 processeurs VLX, deux fois plus rapides que le TXP. Chaque processeur disposait des mêmes disques que sur la machine que nous avons testée. Le résultat de leur benchmark donnait comme valeur limite du throughput assurant un temps de réponse inférieur à deux secondes, 208 TPS [4], ce qui est approximativement 64 fois supérieur à notre résultat (64 car les processeurs sont deux fois plus puissants et 32 fois plus nombreux). Cela tendrait à confirmer l'hypothèse de la linéarité de l'accroissement des performances par rapport à un accroissement du matériel installé.

Qu'est-ce qui permet d'expliquer la différence de performances entre le Vax et le Tandem? C'est ce que nous allons essayer de mettre en lumière maintenant. Notre analyse se fera en parallèle pour les deux machines.

8.3.2. Interprétation des résultats.

a) La conception des systèmes.

Une première constatation est que l'application que nous avons fait tourner est typiquement une application OLTP (OnLine Transaction Processing). Le Tandem est un ordinateur conçu pour l'OLTP. Il est pourvu de différentes fonctionnalités qui permettent de développer, de simuler et d'optimiser des applications OLTP. Le Vax quant à lui n'a pas cette caractéristique. Il s'agit plutôt d'un general-purpose system. Le programmeur n'a pas à sa disposition tous les outils disponibles sur le Tandem et qui permettent d'améliorer sensiblement les performances d'une application OLTP.

b) La simulation des terminaux.

Un second facteur qui influence aussi les performances est, comme nous l'avons déjà dit, la manière dont l'activité du système est simulée. Sur le

Tandem, nous avions à notre disposition un simulateur de terminaux qui permettait d'enregistrer les transactions réalisées à partir d'un terminal et, en utilisant le script enregistré, de simuler l'activité de plusieurs terminaux réalisant les mêmes transactions. Cet outil envoie donc les requêtes au serveur de transaction, qui les exécute. Ces requêtes sont placées dans une file d'attente. Une seule version du serveur de transaction tourne réellement sur le système.

Sur le Vax, la manière de procéder est très différente: nous n'avons pas simulé les terminaux, nous avons dû les faire travailler réellement, c'est-à-dire, faire tourner sur chacun d'eux l'application que nous avions développée. Chaque terminal avait donc son propre serveur de transactions en activité, bien que les fichiers constituant la base de données étaient communs à tous les terminaux. Cette seconde méthode est plus lourde que la première.

c) La charge du système.

Dans les deux cas, nous avons fait tourner notre application sur des systèmes 'vides', c'est-à-dire avec une charge minimale. Si sur le Vax, il n'y avait aucun autre utilisateur en activité que nous au moment où nous menions les tests, sur le Tandem, la charge était non-nulle, quoique très réduite, car certaines applications de la firme ne pouvaient être arrêtées. Cependant, cette différence dans la charge du système ne semble pas avoir été assez importante pour influencer de manière déterminante les performances des deux systèmes.

d) L'architecture de l'application.

Une différence importante dans l'architecture des applications de test a influencé les résultats.

Sur le Tandem, l'application était conçue de la manière suivante:

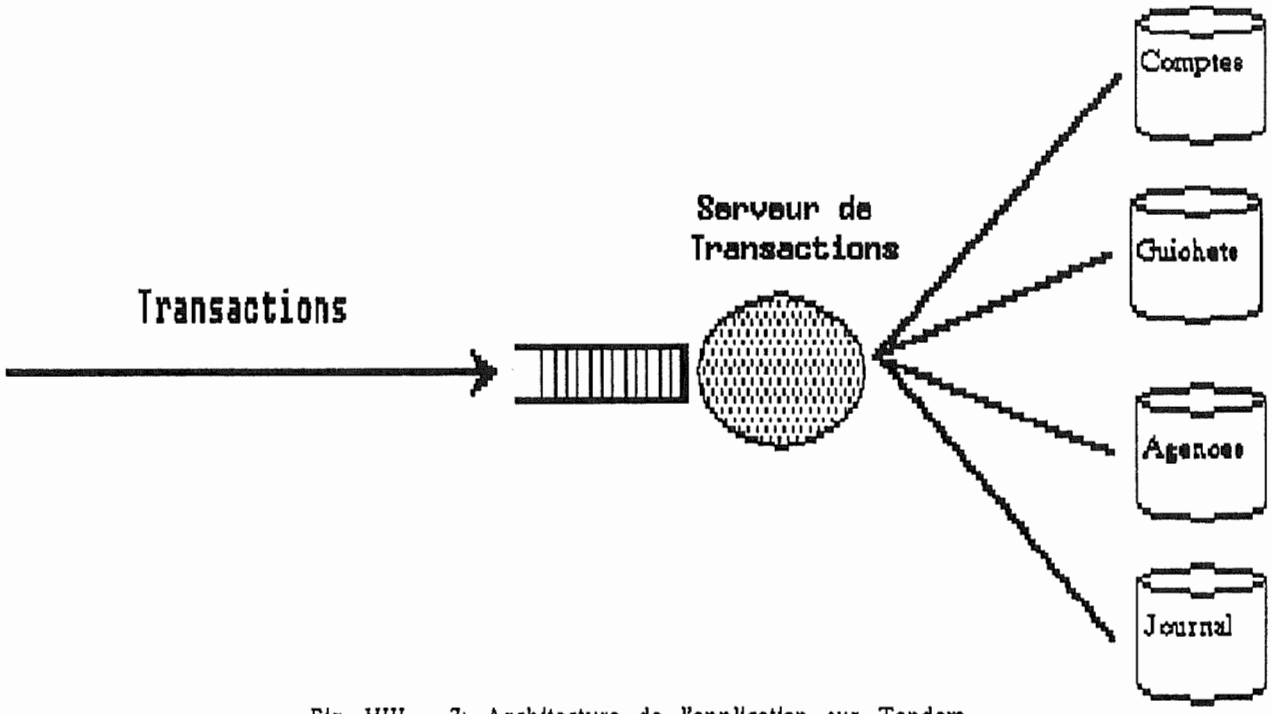


Fig. VIII - 7: Architecture de l'application sur Tandem

Les transactions sont mises dans une file d'attente, devant le serveur. Elles y restent jusqu'au moment où elles sont traitées. Le seul temps d'attente est donc celui qui précède l'exécution de la transaction par le serveur. Le blocage des records dans les fichiers à mettre à jour n'intervient pas, car les transactions sont traitées séquentiellement, en une fois. Il n'y pas de risque que l'exécution d'une transaction soit interrompue par l'exécution d'une autre. Ceci permet un traitement homogène des transactions, grâce au moniteur de transactions.

Sur le Vax, l'application était conçue de manière différente, car nous ne disposions pas d'une architecture requester-server semblable à celle du Tandem, comme exposé au chapitre 5. Voici le chemin suivi par une transaction:

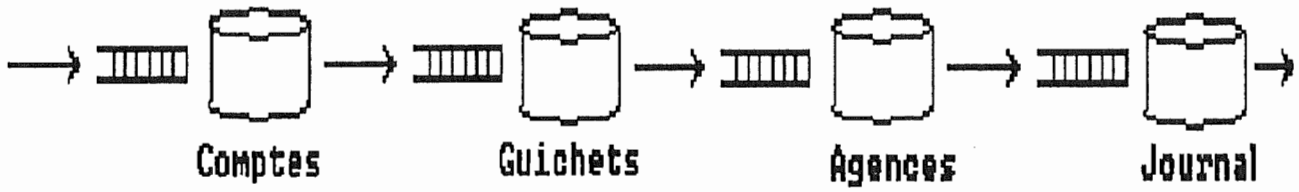


Fig. VIII · 8: Architecture de l'application sur Vax

C'est le même programme qui doit se charger de faire les différentes mises-à-jour, et comme le même programme tourne en différents exemplaires pour différents terminaux, le blocage des records est crucial dans cette conception de l'application, car le programme fait ses mises-à-jour séquentiellement, mais pas nécessairement en un bloc. Il se peut que l'application tournant pour un terminal attende un record qui est utilisé par un autre. Si c'est le cas, le processus se met en attente pour un temps aléatoire avant de répéter la demande d'accès.

Cela signifie qu'une transaction à partir d'un terminal peut être interrompue, voire bloquée par une transaction lancée depuis un autre terminal, et ceci pour un délai assez long. On comprend aisément qu'une telle organisation fait augmenter considérablement le temps de réponse du système.

De plus, on voit qu'il y a quatre files d'attente; même si le temps d'attente individuel à chacune des files est moins important que le temps d'attente à la file du Tandem, la somme de ces temps individuels dépasse le temps d'attente en vigueur sur le Tandem. C'est le problème posé par la décentralisation du traitement, sur les mêmes données. Dans le cas du Tandem, le traitement est centralisé, car un seul programme tourne, et les données sont uniques, les fichiers n'existant qu'en un seul exemplaire. Sur le Vax, par contre, le traitement est décentralisé, car ce sont plusieurs programmes concurrents qui réalisent la mise-à-jour de la base de données; les données sont uniques, comme dans le cas du Tandem.

e) L'architecture des données.

Nous allons examiner les différences dans l'organisation des fichiers, sur les deux machines, et voir en quoi ces différences peuvent donner lieu à des performances divergentes. Parallèlement, nous essaierons d'estimer le temps que met une transaction à être exécutée, sur l'une et l'autre des machines. Ceci nous conduira à un modèle théorique de l'exécution du test.

Bien que l'organisation des fichiers soit la même sur les deux machines (trois fichiers organisés en séquentiel-indexé et un fichier en séquentiel), et que les opérations exécutées dans les deux cas soient identiques (une mise-à-jour par fichier séquentiel-indexé, et l'ajout d'un article au bout du fichier séquentiel), la taille des fichiers est différente. Les fichiers créés sur le Vax ont une taille logique, en nombre d'articles, deux fois moindre que les fichiers créés sur le Tandem, ceci pour des raisons d'espace-disponible.

Nous allons d'abord traiter le cas des fichiers séquentiel-indexés. Nous nous baserons pour ce faire, sur le raisonnement développé dans l'article [14].

Le tableau qui suit présente pour chaque fichier, les caractéristiques propres à chaque système, et celles qui leur sont communes.

Fichier	Nb articles	Longueur article (B)	Longueur de la clé (B)	Blocs de données	Niveaux d'index
	Tandem				
Compte	1000000	100 B	10 B	250000	4
Guichet	10000	100 B	4 B	2500	2
Agence	1000	100 B	3 B	250	2
	Vax				
Compte	500000	100 B	10 B	23256	3
Guichet	5000	100 B	4 B	240	1
Agence	500	100 B	3 B	24	1

Tab. VIII - 14: Structure des fichiers

e.1) Modèlisation du temps d'accès.

Soit un fichier contenant N articles, d'une longueur L_a contenant une clé de longueur L_c. Ce fichier est organisé en blocs, sans chevauchement des articles d'un bloc à l'autre.Supposons que le fichier est organisé avec trois niveaux d'index, numérotés de 1 à 3, le niveau 3 étant le master index. Les données se trouvent au niveau 0. Un bloc peut contenir N_i records d'index ou N_d records de données. Le temps de lecture d'un bloc sur disque est égal à T_{lb}. Le schéma d'accès à un article sera le suivant:

Etape 1: Lire le master index (niveau 3).

Temps nécessaire: T_{lb}

Etape 2: Parcourir le master index pour trouver l'adresse du bloc d'index de niveau 2.

Temps nécessaire: $1/2 * Ni * T_c$

où Tc est le temps nécessaire à la comparaison de la valeur de clé et de la valeur lue dans le master index.

Etape 3: Lire le bloc d'index de niveau 2.

Temps nécessaire: T_{lb}

Etape 4: Parcourir le bloc lu pour trouver l'adresse du bloc d'index de niveau 3.

Temps nécessaire: $1/2 * N_i * T_c$

Etape 5: Lire le bloc d'index de niveau 1.

Temps nécessaire: T_{lb}

Etape 6: Parcourir le bloc lu pour trouver l'adresse du bloc de données cherché.

Temps nécessaire: $1/2 * N_i * T_c$

Etape 7: Lire le bloc de données (niveau 0).

Temps nécessaire T_{lb}

A la fin de l'étape 7, le bloc de données désiré est chargé en mémoire.

Etape 8: Une fois le bloc chargé, il faut encore aller chercher l'article désiré; le temps nécessaire à cette opération est égal à l'expression suivante:

$1/2 * N_d * T_c$

Le coût total (C_t) de chargement en mémoire d'un bloc de données et du positionnement sur l'article voulu, dans un fichier indexé, avec trois niveaux d'index, est donc:

$$C_t = (4 * T_{lb}) + 3 * (1/2 * N_i * T_c) + (1/2 * N_d * T_c)$$

$$= (4 * Tlb) + Tc * ((3/2 * Ni) + (1/2 * Nd))$$

Notons que si les index sont organisés sous forme d'arbre binaire, l'expression ci-dessus devient:

$$C_t = (4 * T_{lb}) + 3*(Log2(N_i*T_c) + (Log2(N_d*T_c)$$

Nous allons maintenant instancier ce raisonnement aux cas particuliers du Tandem et du Vax.

e.2) Instanciation du modèle au cas du Tandem.

Nous traiterons pour l'exemple, le cas du fichier des comptes. Il est organisé en 4 niveaux d'index, les blocs sont dans ce cas-ci de 512 bytes; les disques ont une vitesse d'accès (délai rotationnel et seek time) de 24.8 ms, et une vitesse de transfert de 1.860 MB/s.

Nous pouvons donc estimer le temps moyen de transfert d'un bloc, T_{lb} par l'expression suivante:

$$T_{lb} = \frac{512\ b}{1860\ b/ms} + 24.8\ ms$$

$$= 25.07\ ms$$

Il y a cinq blocs à charger (quatre blocs d'index et un bloc de données), le coût total de chargement sera égal à l'expression suivante:

$$C_{lc} = 5 * 25.07\ ms$$
$$= 125.35\ ms$$

Les index ne sont pas organisés sous forme d'arbre binaire; le calcul du temps nécessaire à la recherche du bloc de niveau inférieur est donc réalisé par:

$$T_r = (1/2 * N_l * T_c)$$

Sur le Tandem, les blocs contiennent un nombre différent de records d'index, selon le niveau d'index que l'on examine. Au niveau 4, il y a 8 records d'index par bloc, au niveau 3, il y en a 31, ainsi qu'aux niveaux 2 et 1. Le coût de recherche dans les blocs d'index sera donc égal à l'expression suivante:

$$C_{ri} = (0.5*8*T_c) + 3*(0.5*31*T_c)$$
$$= 50.4 * Tc$$

Il faut encore calculer le temps nécessaire à la recherche du bon article dans le bloc de données. Un bloc de données contenant 4 articles, le temps nécessaire à cette opération peut être estimé comme suit:

$$C_{rd} = (0.5*4*Tc)$$
$$= 2 * Tc$$

Le coût total de recherche sera donc égal à:

$$C_{tr} = C_{ri} + C_{rd}$$
$$= 50.4 * T_c + 2 * T_c$$
$$= 52.4 * T_c$$

Dès lors, le coût total d'accès à un article dans le fichier des comptes, organisé comme spécifié ci-dessus sera égal à l'expression suivante:

$$C_t = C_{tr} + C_{tc}$$
$$= 125.35 \text{ ms} + 52.4 * T_c$$

Dans l'article [14], une estimation de T_c est fournie, sur base de calculs statistiques réalisés par une firme. Cette estimation donne Tc = 0.0015 sec. Une estimation du coût total serait donc:

$$Ct = 125.35 \text{ ms} + 52.4 * 1.5 \text{ ms}$$
$$= 203.95 \text{ ms}$$

Nous avons calculé jusqu'à présent le temps nécessaire au chargement de l'article à mettre à jour. Il faut encore exécuter cette mise-à-jour, et ré-écrire l'article sur le disque.

La ré-écriture de l'article sur le disque ne nécessite que l'écriture d'un bloc, les blocs d'index n'étant pas modifiés. Le coût d'écriture peut être estimé par l'expression suivante, si l'on admet l'hypothèse que le temps d'écriture est égal au temps de lecture:

$$C_e = 24.8 \text{ ms} + \frac{512 \text{ B}}{1860 \text{ B/s}}$$
$$= 25.07 \text{ ms}$$

Pour le temps nécessaire à la mise-à-jour de l'article, nous ne disposions pas de mesures qui nous permettent de l'estimer.

Le temps d'activité I/O total peut être estimé par le calcul suivant:

$$C_{i/o} = C_l + C_e$$
$$= 203.95\text{ms} + 25.07 \text{ ms}$$
$$= 229.02 \text{ ms}$$

Le tableau qui suit présente le résultat des calculs réalisés pour les fichiers agence et guichet:

Fichier	Temps total de chargement	Temps total de mise-à-jour
Compte	203.95 ms	229.02 ms
Agence	119.46 ms	144.53 ms
Guichet	153.21 ms	178.28 ms

Tab. VIII · 15: Temps total de mise-à-jour.

Le temps total passé à la mise-à-jour des fichiers indexés peut donc être estimé par l'expression:

$$C_{ind} = 229.02 \text{ ms} + 178.28 \text{ ms} + 144.53 \text{ ms}$$

$$= 551.83 \text{ ms}$$

e.3) Instanciation du modèle au cas du Vax.

Nous traiterons en détail le cas du fichier des comptes. Celui-ci est organisé en 3 niveaux d'index. Les articles sont rangés dans des blocs de 512 bytes, mais l'unité de transfert entre le disque et la mémoire est de un bucket, soit deux blocs. Les disques ont une vitesse d'accès (délai rotationnel et seek time) de 36.3 ms, et une vitesse de transfert de 2.200 MB/s. Les articles peuvent être compressés de manière très importante; en effet, un caractère qui apparaît plus de 5 fois consécutivement dans un record est automatiquement réduit à un seul.

Le temps moyen de transfert d'un bucket, T_{lb}, est estimé dans l'expression suivante:

$$T_{lb} = \frac{1024 \text{ b}}{2200 \text{ B/ms}} + 36.3 \text{ ms}$$

$$= 36.77 \text{ ms}$$

Il y a, dans la recherche d'un bloc de données, trois blocs d'index plus un bloc de données à charger, soit quatre blocs. Le coût total de chargement sera donc:

$$C_{tc} = 4 * 36.77 \text{ ms}$$

$$= 147.08 \text{ ms}$$

Ici non plus, les index ne sont pas organisés sous la forme d'arbres binaires; le coût total de comparaison d'une clé avec les clés d'un bloc, pour trouver l'adresse du bloc de niveau inférieur est donc estimé par l'expression:

$$T_r = (1/2 * N_i * T_c)$$

Sur le Vax, les blocs des niveaux d'index 1 et 2 contiennent 38 records, tandis que les blocs du niveau 3 n'en contiennent que 2. On en déduit que, comme les blocs sont regroupés en buckets de 1024 bytes, le coût de recherche dans les blocs d'index, pour trouver l'adresse du bloc de niveau inférieur sera égal à l'expression suivante:

$$\begin{aligned} C_{ri} &= (0.5*4*T_c) + 2*(0.5*76*T_c) \\ &= 78 * T_c \end{aligned}$$

Nous utiliserons l'estimation de T_c donnée dans [14], soit T_c = 1.5 ms, ce qui donne le résultat suivant:

$$\begin{aligned} C_{ri} &= (78 * 1.5 \text{ ms}) \\ &= 117 \text{ ms} \end{aligned}$$

Il faut encore calculer le temps nécessaire à la recherche du bon article dans le bloc de donnée chargé; un bloc de données contient sur le Vax, 21 articles. Le temps nécessaire à la sélection du bon article sera en moyenne égal à l'expression suivante:

$$\begin{aligned} C_{rd} &= (0.5*42*T_c) \\ &= 21 * T_c \\ &= 31.5 \text{ ms} \end{aligned}$$

On peut estimer le coût total de recherche par:

$$\begin{aligned} C_{tr} &= C_{rl} + C_{rd} \\ &= 117 \text{ ms} + 31.5 \text{ ms} \\ &= 148.5 \text{ ms} \end{aligned}$$

Le coût total de chargement d'un article dans le fichier des comptes, organisé de la manière spécifiée ci-dessus, peut être estimé de la manière suivante:

$$\begin{aligned} C_t &= C_{tr} + C_{tc} \\ &= 148.5 \text{ ms} + 147.08 \text{ ms} \\ &= 295.58 \text{ ms} \end{aligned}$$

Il faut rajouter à ce temps, le temps nécessaire à l'écriture de l'article mis-à-jour. Cette opération ne nécessite que l'écriture d'un bloc, car les blocs d'index ne sont pas modifiés. Ce coût peut être estimé de la manière suivante, si l'on admet l'hypothèse selon laquelle le coût d'une lecture est égal au coût d'une écriture:

$$\begin{aligned} C_e &= 36.3 \text{ ms} + \frac{1024 \text{ B}}{2200 \text{ B/s}} \\ &= 36.7 \text{ ms} \end{aligned}$$

Nous ne disposons malheureusement pas de mesures qui nous auraient permis d'estimer le temps CPU nécessité par une telle opération.

Le temps I/O total peut être estimé par l'expression suivante:

$$C_{I/o} = C_t + C_e$$
$$= 295.58 + 36.7 \text{ ms}$$
$$= 332.28 \text{ ms}$$

Le tableau qui suit présente les résultats des mêmes calculs pour les fichiers agence et guichet:

Fichier	Temps total de chargement	Temps total de mise-à-jour
Compte	332.3 ms	369 ms
Agence	128.4 ms	165.1 ms
Guichet	194.9 ms	231.6 ms

Tab. VIII - 16: Temps total de mise-à-jour.

Le temps total passé en I/O pour la mise-à-jour des fichiers indexés peut être estimé par l'expression suivante:

$$C_{Ind} = 332.28 \text{ ms} + 228.4 \text{ ms} + 194.9 \text{ ms}$$
$$= 755.58 \text{ ms}$$

e.4) Réflexion sur l'organisation des fichiers indexés.

En bref, nous pouvons expliquer les différences de performance des deux systèmes, dans le traitement des fichiers indexés, par différents facteurs.

D'abord, on peut voir que sur le Tandem, il y a plus de niveaux d'index que sur le Vax. Cela signifie que le nombre d'accès physiques au disque sera plus élevé dans le premier cas que dans le second.

Ensuite, il y a la vitesse des disques, qui globalement, est plus élevée sur Tandem que sur Vax, ce qui permet des transferts plus rapides. La taille des blocs est plus élevée sur le Vax, que sur le Tandem, ce qui permet d'y ranger plus d'articles. Cependant, du fait que, dans l'organisation séquentielle-indexée, le nombre de blocs à charger reste fixe, cela n'améliorait pas les performances du Vax, d'autant moins qu'un bloc contenant plus d'articles, les coûts de comparaison de la clé de recherche avec les clés dans les blocs d'index s'en trouvaient accrus d'autant. Cette remarque est encore plus pertinente quand l'on considère le fait que les articles, sur Vax étaient comprimés, et que cette compression permettait d'économiser jusqu'à parfois 90 % de la place prise par un article. Plus d'articles pouvant être rangés dans un bloc, les coûts de recherche d'un article dans un bloc sont aussi plus élevés.

e.5) Le cas du fichier séquentiel.

Nous allons maintenant traiter le cas du fichier journal, qui est un fichier séquentiel auquel chaque transaction vient ajouter un article. Il est clair qu'à chaque écriture d'un article, tout le fichier n'est pas relu. Sur chacun des deux ordinateurs testés, le système de gestion de fichiers connait l'adresse du dernier bloc du fichier. Pour insérer un nouvel article, il charge le dernier bloc. Selon l'état de ce bloc, deux cas peuvent se présenter: soit ce bloc n'est pas rempli, et dans ce cas on y ajoute le nouvel article et le recopie sur le disque, soit il est rempli, et on crée un nouveau bloc, on y range le nouvel article, et on écrit ce nouveau bloc sur disque. Si un bloc peut contenir N_b articles, on devra créer un nouveau bloc, en moyenne $(1/N_b)$ fois. Si l'on admet l'hypothèse que le coût d'écriture d'un bloc est égal au coût de lecture d'un bloc, une estimation du coût d'écriture du nouvel article pourra être calculée par l'expression suivante:

$$C_{en} = \left(\frac{N_b - 1}{N_b} * 2 * C_{lb}\right) + \left(\frac{1}{N_b} * ((2 * C_{lb}) + C_{cb})\right)$$

où C_{cb} = le coût de création d'un bloc

C_{lb} = le coût de lecture d'un bloc

$$= \text{Temps d'accès} + \frac{\text{Taille d'un bloc}}{\text{Vitesse de transfert}}$$

Si l'on adapte cette formule au cas du Tandem, on obtient l'expression suivante:

$$C_{lb} = 24.8 \text{ ms} + \frac{1024 \text{ B}}{1860 \text{ MB/s}}$$

$$= 25.3 \text{ ms}$$

$$C_{en} = \frac{18}{19} * (2 * 25.3 \text{ ms}) + \frac{1}{19} * ((2 * 25.3 \text{ ms}) + C_{cb})$$

$$= 47.9 \text{ ms} + 2.6 \text{ ms} + \frac{C_{cb}}{19}$$

Nous n'avons pas trouvé d'estimation pour C_{cb}, mais nous pensons que c'est une mesure assez petite. Si nous la divisons encore par 19, cela devient insignifiant. Nous approcherons donc C_{en} de la manière suivante:

$$C_{en} = 47.9 \text{ ms} + 2.6 \text{ ms}$$
$$= 50.5 \text{ ms}$$

Il ressort de ce résultat et de celui calculé au point e.2 que le temps total nécessité par la mise-à-jour des fichiers par une transaction peut être calculé comme suit:

$$\begin{aligned} C_t &= C_{ind} + C_{en} \\ &= 551.83 \text{ ms} + 50.5 \text{ ms} \\ &= 602.33 \text{ ms} \end{aligned}$$

Si l'on suit le même raisonnement pour le Vax, on obtient:

$$C_{lb} = 36.3 \text{ ms} + \frac{512 \text{ B}}{2200 \text{ MB/s}}$$

$$= 36.5 \text{ ms}$$

$$C_{en} = \frac{10}{11} * (2 * 36.5 \text{ ms}) + \frac{1}{11} * ((2 * 36.5 \text{ ms}) + C_{cb}))$$

$$= 66.3 \text{ ms} + 6.6 \text{ ms} + \frac{C_{cb}}{11}$$

Si l'on 'oublie' le dernier terme, on approche C_{en} par:

$$\begin{aligned} C_{en} &= 66.3 \text{ ms} + 6.6 \text{ ms} \\ &= 72.9 \text{ ms} \end{aligned}$$

Si l'on combine ce résultat et celui obtenu au point e.3, on obtient une estimation du temps total de mise-à-jour des fichiers pour une transaction:

$$\begin{aligned} C_t &= C_{ind} + C_{en} \\ &= 755.58 \text{ ms} + 72.9 \text{ ms} \\ &= 828.48 \text{ ms} \end{aligned}$$

Une estimation du coût total de traitement d'une transaction est donc donnée, pour les deux ordinateurs, par:

C_{Tandem} = 602.33 ms
C_{Vax} = 828.48 ms

8.3.3. Conclusion.

Des points évoqués ci-dessus, on déduit que le Vax traitera une transaction nettement moins rapidement que le Tandem, et dans des conditions de traitement plus défavorables; on peut en déduire que le système sera plus vite engorgé, comme nous l'avons constaté lors de nos mesures. Cependant, il faut rappeler que le Tandem est conçu pour le traitement de transactions, ce qui n'est pas le cas du Vax. Cela explique que pour une application de taille modeste, le Vax offrira des performances raisonnables, mais dès que le volume de données à traiter ou que la complexité de l'application ou de ses conditions d'exécution augmente, les performances se dégradent très rapidement. Par contre, le Tandem, dont toute l'architecture, tant hardware que software, est étudiée pour le traitement de transactions, offre des performances à la mesure des moyens mis à la disposition du développeur, c'est-à-dire nettement supérieures à celles du Vax.

9. Conclusion.

9.0. Introduction.

Nous voudrions placer la conclusion de ce travail sur différents plans: premièrement, nous aimerions, après avoir étudié et approfondi le concept des tests ET1, après en avoir implémenté deux, de deux manières différentes, après avoir collecté et interprété les résultats des prises de mesures réalisées lors de l'exécution des tests, il nous paraît important de voir dans quelle mesure les objectifs fixés par les créateurs des tests ET1 ont été atteints.

Ensuite, nous aimerions voir dans quelle mesure les résultats fournis par les tests que nous avons menés, nous permettent d'aboutir à une conclusion significative, d'un point de vue des performances des deux systèmes.

9.1. Le concept ET1.

L'idée sous-jacente à la création des tests ET1 était d'étudier des aspects des systèmes informatiques qui étaient passés sous silence par la plupart des benchmarks précédents. Ces aspects étaient plutôt de l'ordre de l'utilisateur final, ou du programmeur moyen, travaillant sur un ordinateur.

Il nous semble que certains de ces aspects sont bien mis en lumière par l'implémentation et l'exécution de tests ET1. Mais nous pensons que les aspects testés restent encore très techniques et fort limités à une partie du système testé, et pas aussi proches des préoccupations de l'utilisateur final du système que ce que les pères des ET1 l'auraient voulu. En effet, qu'il s'agisse de l'architecture I/O physique ou de l'architecture I/O logique du système, éprouvées respectivement par le test Sort et le test Scan, ces préoccupations nous semblent encore loin de celles de l'utilisateur final. En effet, tous les utilisateurs d'un système ne sont pas conscients de la manière dont le tri

fonctionne, ou dont les disques sont organisés. Ce sont des aspects relevant d'un niveau plus technique. Le test de la transaction bancaire est peut être un peu plus parlant, car les aspects testés relèvent plus des outils software mis à la disposition d'un utilisateur, ou d'un développeur. Cependant, pour ce qui est de la réalisation des objectifs donnés à ce benchmark par ses créateurs, nous pensons qu'ils sont bien réalisés. En effet, le benchmark de tri éprouve, comme prévu, l'architecture hardware I/O du système, le benchmark de mise-à-jour isole correctement l'aspect software de l'architecture I/O du système, et la transaction bancaire met clairement en évidence les facilités et outils offerts au développeur d'applications pour mener à bien sa tâche.

Il nous semble encore que la définition des tests manque de rigueur et de précision. Comment peut-on donner du sens à des résultats des tests si ceux-ci ont été interprétés différemment? C'est ce qui est développé dans l'article [5]: *'All TP1's are not created equal!'*. Le manque total de littérature et de personnes ayant l'expérience des tests ET1 se fait cruellement sentir à tout qui est interessé par le sujet.

Nous pensons de plus que la définition d'un benchmark n'est pas une chose suffisante pour lui donner du sens. Il faut aussi donner des lignes de conduite quant à l'utilisation des résultats. Si l'on s'en tient à la définition des tests [1,12], après une campagne ET1, on aura obtenu trois résultats: le temps nécessaire au tri d'un fichier de taille donnée, le temps nécessaire à la mise-à-jour d'un fichier de taille donnée et le nombre de transactions pouvant être traitées par seconde, de telle manière que le temps de réponse reste inférieur à un temps donné. Il nous semble que ces résultats sont en eux-même insuffisants pour donner une idée des performances d'une machine. A moins que l'on se contente d'une affirmation telle 'Cet ordinateur est plus performant que tel autre, car son throughput est supérieur', ou 'car il trie le même fichier plus rapidement', la nécessité d'une analyse plus fouillée des causes permettant d'expliquer les différences de performances mesurées devient réelle; les résultats décrits dans le standard nous paraîssent se situer à un niveau trop microscopique, et nous pensons que pour une bonne compréhension du benchmark et surtout du système testé, il est important de se placer à un niveau plus macroscopique.

Il faut enfin signaler que le benchmark ET1, dans son ensemble ne paraît pas être employé à grande échelle. D'après nos contacts avec différents

constructeurs, peu de personnes en ont entendu parler, et les cas d'implémentation sont encore plus rares.

Peut être le benchmark ET2, qui d'après certaines sources serait en cours d'élaboration, palliera-t-il aux insuffisances de l'ET1? Nous n'avons pas reçu d'information à ce sujet, pour permettre de répondre à cette question.

9.2. Les tests menés.

La question posée est la suivante: 'Le Vax est-il meilleur que le Tandem, ou le Tandem lui est-il supérieur?'. A notre sens, il n'est pas possible de conclure dans l'un ou l'autre sens; notre conclusion est plus nuancée.

Il nous semble que le Vax a une architecture I/O physique sensiblement semblable à celle du Tandem, d'après les résultats obtenus lors du benchmark de tri qui sont du même ordre sur l'un ou l'autre système. Les disques ont une vitesse de transfert comparable, bien que le temps d'accès moyen à un secteur soit plus rapide sur le Tandem que sur le Vax. Le Vax compense cette différence par une unité de transfert I/O plus importante que le Tandem. Le benchmark de tri part de l'hypothèse que le processus de tri implémenté sur une machine est conçu de manière à supprimer tout l'overhead software, pour ne garder comme mesures, que les performances brutes de l'architecture hardware I/O.

La conclusion du test Scan est différente; les résultats obtenus indiquent que l'architecture software du Vax est plus performante que celle du Tandem. Nous avons justifié cette conclusion dans le chapitre 8. Rappelons simplement que le facteur de différence le plus important est le nombre de blocs physiques chargés en même temps, qui est 16 fois plus élevé sur le Vax que sur le Tandem, ce qui permet de réduire considérablement le temps de lecture/écriture sur le disque, principale activité du test Scan.

Pour ce qui est de la transaction bancaire, une conclusion de comparaison est difficile à tirer, car le Vax n'entre pas, à notre avis, dans le domaine de l'OnLine Transaction Processing. Il n'offre pas les fonctionnalités, sinon indispensables, du moins très utiles au développement et à l'exécution

d'applications OLTP; ces fonctionnalités sont offertes sur le Tandem. Cependant, il faut rappeler que l'objectif de la transaction bancaire est précisément de tester la capacité du système à traiter des transactions, ainsi que les services offerts par le système pour de telles applications. Le Vax ne possède pas de tels services, ce qui ne permet pas d'obtenir des programmes aussi performants que sur le Tandem.

Comme nous l'avons dit plus haut, nous avons dû faire de nombreuses concessions au standard pour pouvoir adapter ce benchmark sur le Vax. Cela se ressent dans les résultats. Nous pensons que les moyens nécessaires à l'implémentation complète d'un benchmark ET1 doivent être considérables. Le benchmark original avait été implémenté sur deux ordinateurs, de telle manière que les transactions étaient générées par l'un des deux ordinateurs et traitées par l'autre, ceci pour isoler correctement les performances du système testé. L'ordinateur qui traitait les transactions était composés de 32 processeurs, munis de quatre disques chacun. Nous ne disposions évidemment pas d'un tel équipement.

Ces points nous empêchent de tirer une conclusion de comparaison pour la transaction bancaire. Si l'on s'en tient aux résultats observés, nous ne pouvons dire qu'une chose, c'est que le Tandem traite rapidement des transactions, alors que les performances du Vax se dégradent très rapidement dès que la quantité de données à traiter devient plus importante.

Tout ceci nous conduit à la conclusion suivante: les deux systèmes sont prévus pour des usages différents; le Tandem est entièrement conçu pour gérer des applications OLTP et pour résister à une charge importante sur le système. En conséquence, il traite aussi bien un volume important de données qu'un volume plus réduit. Le Vax, quant à lui, semble prévu pour supporter toutes les formes d'utilisation, mais de manière réduite. Dès que les données à traiter deviennent plus volumineuses, les performances se dégradent rapidement.

Une véritable comparaison entre les deux ordinateurs ne pouvait donc se faire qu'au niveau d'éléments particuliers du système, et non au niveau plus global du système lui-même.

Références bibliographiques

[1]

Anon et al. "A Measure of transaction processing power" Datamation, April 1, 1985, pp112-118

[2]

Tandem Computers, "NonStop SQL Performance", NonStop SQL Benchmark Report, February 1987

[3]

Vijay Trehan, "Debit Credit Benchmark implementation guidelines", Internal Digital Memo, March 28, 1986

[4]

Tandem Computers Incorporated, "Workbook of the Topgun Benchmark Demonstrating NonStop SQL Performance on 32 Tandem VLX processors and audited by Codd and Date Consulting Group", March 6, 1987

[5]

Steven Caniano, "All TP1's are not created equal" Datamation August 15, 1988 pp 51-53

[6]

Tandem Computers Incorporated, "Fastsort: An External Sort Using Parallel Processing", Tandem Computer Manual

[7]

Gray J., Gawlick D. "One Thousand Transaction Per Second" Proceedings of IEEE COMPCON, San Francisco, IEEE Press, Feb 1985

[8]

Gawlick D., "Processing of the Hot Spots in Database Systems", Proceedings of IEEE COMPCON, San Francisco, IEEE Press, Feb. 1985.

[9]

Bitton, Dewitt, Turbyfill, "Benchmarking Database systems: a systematic approach", in Proceedings VLDB, 1983 (Florence).

[10]

Gray J., "Notes on Databases Operating Systems", pp. 395-396, in Lecture Notes in Computer Science, Vol. 60, Bayer-Seegmuller eds., Springer Verlag, 1978.

[11]

Park S. Miller K., "Random Number Generators: Good ones are hard to find", Communications of the ACM, October 1988, Vol. 31 Nb 10.

[12]

Tandem Technical Report 85.2.

[13]

Knuth "The art of Computer Programming: Sorting and Searching"

[14]

Fedorowicz D.," Database Performance Evaluation in an Indexed File Environment", ACM Transactions on Database Systems, Vol. 12, No. 1 March 1987, pp. 85-110.

[15]

Digital Equipment Corporation, "VAX/VMS System Software Handbook", 1987.

[16]

Digital Equipment Corporation, "VAX Architecture Handbook", 1987.

[17]

Digital Equipment Corporation, "VAX Systems Hardware Handbook", 1987

[18]

Hainaut J.-L., "Technologie des fichiers", Cours donné en 1986.

[19]

Hainaut J.-L., "Conception assistée des applications informatiques; 2. conception de la base de données.", Presses Universitaires de Namur, 1986.

[20]

Lesuisse R., "Programmation d'applications de gestion en Cobol", Cours donné en 1986.

Annexes.

Les pages qui suivent présentent le listing des programmes utilisés pour l'implémentation des tests ET1. Voici la tâche réalisée par chacun des programmes.

Et1scan	Programme implémentant le test Scan sur Tandem et sur Vax
Et1sort	Programme implémentant le test Sort sur Tandem et sur Vax.
Et1req	Programme implémentant le requester de la transaction bancaire sur Tandem.
Et1serv	Programme implémentant le serveur de la transaction bancaire sur Tandem.
Debitcredit	Programme implémentant la transaction bancaire sur Vax.
Bdsort	Programme de chargement du fichier utilisé par les tests Scan et Sort sur Tandem et sur Vax.
Bdjournal	Programme de chargement du fichier *journal* de la transaction bancaire sur Tandem et sur Vax.
Bdagence	Programme de chargement du fichier *agence* de la transaction bancaire sur Tandem et sur Vax.
Bdguichet	Programme de chargement du fichier *guichet* de la transaction bancaire sur Tandem et sur Vax.
Bdcompte	Programme de chargement du fichier *compte* de la transaction bancaire sur Tandem et sur Vax.

Après le listing de ces programmes se trouvent les spécifications complètes des fichiers utilisés sur le Tandem. Nous avons renoncé à annexer tous les listings des mesures prises, car étant donné la quantité et le format de ces informations, il nous était impossible de les relier

```
Identification division.
Program-id.  et1scan.

Environment division.

Input-Output section.
File Control.
     Select fich-entree assign to "entree"
     organization is sequential
     access mode is sequential.

Data division.

File Section.
  FD fich-entree.
     01 f-art.
     02 f-champ1 pic 9(10).
     02 f-champ2 pic x(90).
Working-storage section.
  01  eof-fichier pic 9.
  01 w-art.
     02 w-champ1 pic 9(10).
     02 w-champ2 pic x(90).

Procedure division.

P1 section.
 main-loop.
     perform p2-ouverture-fichier.
     perform p2-main 1000 times.
     perform p2-fermeture-fichier.
     stop run.

P2 section.
 p2-main.
     enter tal begintransaction.
     perform p2-scan 1000 times.
     if eof-fichier = 0 enter tal endtransaction.
 p2-ouverture-fichier.
     open i-o fich-entree shared.
     move 0 to eof-fichier.
 p2-fermeture-fichier.
     close fich-entree.
 p2-scan.
     if eof-fichier = 1 perform p3-erreur.
     perform p3-lire-art.
     perform p3-maj.

P3 section.
 p3-lire-art.
     read fich-entree with lock
               at end move 1 to eof-fichier.
     move f-art to w-art.
 p3-maj.
     compute w-champ1 = w-champ1 + 10.
     rewrite f-art from w-art with unlock.
 p3-erreur.
     enter tal aborttransaction.
```

```
Identification division.
Program-id. et1sort.

Environment division.

Input-Output section.
File Control.
     Select fich-entree assign to "entree"
     organization is sequential
     access mode is sequential.
     Select fich-sortie assign to "sortie"
     organization is sequential
     access mode is sequential.
     Select fich-tri assign to "fichtri".

Data division.

  File section.
     FD fich-entree.
     01 fiche pic x(100).
     FD fich-sortie.
     01 fichs pic x(100).
     SD fich-tri.
     01 art-tri.
        02 cle pic 9(10).
        02 filler pic x(90).

Procedure division.

Tri section.
     Sort fich-tri on ascending key cle
     using fich-entree
     giving fich-sortie.
     Stop run.
```

```
Identification division.
Program-id. et1req.

Environment division.
Configuration section.
 source-computer. Tandem-NSII/TXP.
 object-computer. Tandem-NSII/TXP,
 terminal is t16-6520.
 special-names.
 F16-key is F16.

Data division.
 Working-storage section.
   01 w-art.
     02 w-code-rep pic s9(4).
     02 w-fcode pic 9(3).
     02 w-num-agence pic 9(3).
     02 w-num-guichet pic 9(4).
     02 w-num-compte pic 9(10).
     02 w-montant-debit pic 9(2).
     02 w-nom-cli pic x(15).
     02 w-prenom-cli pic x(15).
     02 w-tel-cli pic x(10).
     02 w-ville-cli pic x(10).
     02 w-comment pic x(24).

   01 w-reply.
     02 w-code-reply pic s9(4).
     02 w-fcode-reply pic 9(3).
     02 w-num-agence-reply pic 9(3).
     02 w-num-guichet-reply pic 9(4).
     02 w-num-compte-reply pic 9(10).
     02 w-montant-debit-reply pic 9(2).
     02 w-nom-reply pic x(15).
     02 w-prenom-reply pic x(15).
     02 w-tel-reply pic x(10).
     02 w-ville-reply pic x(10).
     02 w-comment-reply pic x(24).
     02 filler pic x(100).

screen section.
 01 ecran-debit base size 24, 80.
     02 filler at 1, 35 value "DEBIT D'UN COMPTE".
     02 filler at 2, 35 value "--------------------------
------".
     02 filler at 5, 1 value "No Compte:".
     02 s-num-compte at 5, 12 pic 9(10)
                          from w-num-compte-reply
                          to w-num-compte.

     02 filler at 6, 1 value "Nom du client:".
     02 s-nom-cli at 6, 16 pic x(15)
                          from w-nom-reply
                          to w-nom-cli.
     02 filler at 6, 40 value "Pr nom du client:".
     02 s-prenom-cli at 6, 58 pic x(15)
                          from w-prenom-reply
```

```
                        to w-prenom-cli.
      02 filler at 7, 1 value "Ville:".
      02 s-ville-cli at 7, 8 pic x(10)
                        from w-ville-reply
                        to w-ville-cli.
      02 filler at 7,25 value "Telephone:".
      02 s-tel-cli at 77, 36 pic x(10)
                        from w-tel-reply
                        to w-tel-cli.
      02 filler at 9, 1 value "No de l'agence:".
      02 s-num-agence at 9, 17 pic 9(3)
                        from w-num-agence-reply
                        to w-num-agence.
      02 filler at 9, 25 value "No de guichet:".
      02 s-num-guichet at 9, 40 pic 9(4)
                        from w-num-guichet-reply
                        to w-num-guichet.
      02 filler at 10, 1 value "Montant a debiter:".
      02 s-montant-debit at 10, 20 pic 9(2)
                        from w-montant-debit-reply
                        to w-montant-debit.
      02 filler at 20, 20 value "Comment.".
      02 s-comment at 20, 30 pic x(24)
                        from w-comment-reply
                        to w-comment.

Procedure division.

P0 section.
     perform p2-init.
     perform p1-main-loop until w-nom-cli = "end".
     stop run.

P1 section.
 P1-main-loop.
     begin-transaction.
     perform p2-accept.
     if w-nom-cli = "end" abort-transaction.
     if w-nom-cli not = "end" perform p2-send.
     if w-nom-cli not = "end" end-transaction.
P2 section.
 p2-init.
     move 0 to w-num-agence-reply.
     move 0 to w-num-guichet-reply.
     move 0 to w-num-compte-reply.
     move 0 to w-montant-debit-reply.
     move "" to w-prenom-reply.
     move "" to w-nom-reply.
     move "" to w-tel-reply.
     move "" to w-ville-reply.
     move " Entree des donnees " to w-comment-reply.
     display base ecran-debit.
 p2-accept.
     display ecran-debit.
     accept ecran-debit until F16-key.
 p2-send.
     send w-art to "set1"
          reply code 0, 1 yields w-reply.
```

```
Identification division.
Program-id. etlserv.

Environment division.
Configuration section.
 source-computer. Tandem-NSII/TXP.
 object-computer. Tandem-NSII/TXP.

Input-output section.
  file-control.
     select message-in assign to $receive.
     select message-out assign to $receive.
     select agence assign to "fagence"
          organization is indexed
          record key is f-num-agence
          access mode is random.
     select compte assign to "fcompte"
          organization is indexed
          record key is f-num-compte
          access mode is random.
     select guichet assign to "fguichet"
          organization is indexed
          record key is f-num-guichet
          access mode is random.
     select journal assign to "fjournal"
          organization is sequential
          access mode is sequential.

Data division.
 file section.
   FD agence.
     01 f-art-agence.
     02 f-num-agence pic 9(3).
     02 f-solde-agence pic 9(9).
     02 filler pic x(88).
   FD guichet.
     01 f-art-guichet.
     02 f-num-guichet pic 9(4).
     02 f-solde-guichet pic 9(9).
     02 filler pic x(87).
   FD compte.
     01 f-art-compte.
     02 f-num-compte pic 9(10).
     02 f-solde-compte pic 9(9).
     02 filler pic x(81).
FD journal.
     01 f-art-journal pic x(50).
```

```
FD message-in.
    01 entry-msg.
     02 in-code-rep pic s9(4).
     02 in-fcode pic 9(3).
     02 in-num-agence pic 9(3).
     02 in-num-guichet pic 9(4).
     02 in-num-compte pic 9(10).
     02 in-montant-debit pic 9(2).
     02 in-nom-cli pic x(15).
     02 in-prenom-cli pic x(15).
     02 in-tel-cli pic x(10).
     02 in-ville-cli pic x(10).
     02 in-comment pic x(24).

FD message-out.
    01 out-msg.
     02 out-code-rep pic s9(4).
     02 out-fcode pic 9(3).
     02 out-num-agence pic 9(3).
     02 out-num-guichet pic 9(4).
     02 out-num-compte pic 9(10).
     02 out-montant-debit pic 9(2).
     02 out-nom-cli pic x(15).
     02 out-prenom-cli pic x(15).
     02 out-tel-cli pic x(10).
     02 out-ville-cli pic x(10).
     02 out-comment pic x(24).
     02 filler pic x(100).

Working-storage section.
 01 tr-descr-trans.
     02 tr-num-compte pic 9(10).
     02 tr-num-agence pic 9(3).
     02 tr-num-guichet pic 9(4).
     02 filler pic x(33) value "Descritpion transaction".
 01 w-descr-trans pic x(50).
 01 w-num-compte pic 9(10).
 01 w-num-agence pic 9(3).
 01 w-num-guichet pic 9(4).
 01 w-eof-receive pic 9.

Procedure division.

P0 section.
 loop.
     perform p1-init.
     perform p1-main-loop until w-eof-receive = 1.
     perform p2-term.
     stop run.

p1 section.
 p1-main-loop.
     perform p2-init.
     perform p2-maj.
     perform p2-reply.
```

```
 p1-init.
     open i-o guichet shared.
     open i-o compte shared.
     open i-o agence shared.
     open extend journal shared.
     open input message-in shared.
     open output message-out shared.
     move 0 to w-eof-receive.

p2 section.
 p2-init.
     read message-in at end move 1 to w-eof-receive.
     move in-num-agence to w-num-agence.
     move in-num-guichet to w-num-guichet.
     move in-num-compte to w-num-compte.
 p2-term.
     close agence.
     close journal.
     close guichet.
     close compte.
     close message-in.
     close message-out.
p2-maj.
     move w-num-agence to f-num-agence.
     read agence;
          with lock
          invalid key perform p3-erreur.
     compute f-solde-agence = f-solde-agence + 5.
     rewrite f-art-agence with unlock.
     move w-num-compte to f-num-compte.
     read compte;
          with lock
          invalid key perform p3-erreur.
     compute f-solde-compte = f-solde-compte + 5.
     rewrite f-art-compte with unlock.
     move w-num-guichet to f-num-guichet.
     read guichet;
          with lock
          invalid key perform p3-erreur.
     compute f-solde-guichet = f-solde-guichet + 5.
     rewrite f-art-guichet with unlock.
     move in-num-agence to tr-num-agence.
     move in-num-guichet to tr-num-guichet.
     move in-num-compte to tr-num-compte.
     move tr-descr-trans to f-art-journal.
     write f-art-journal.
     if out-code-rep not = 1
               move 0 to out-code-rep.
     if out-code-rep = 0
               move "Transaction OK" to out-comment.
 p2-reply.
     move 1 to out-fcode.
     move in-num-agence to out-num-agence.
     move in-num-guichet to out-num-guichet.
     move in-num-compte to out-num-compte.
     move f-solde-compte to out-montant-debit.
     move in-num-cli to out-nom-cli.
     move in-prenom-cli to out-prenom-cli.
```

```
           move in-ville-cli to out-ville-cli.
           move in-tel-cli to out-tel-cli.
           write out-msg.

       P3 section.
        p3-erreur.
           move "Bad Datas " to out-comment.
           move 1 to out-code-rep.
```

```
Identification division.
Program-id. debitcredit.

Environment division.
Input-Output section.
  File-control.
     select agence assign to "fagence.ndx"
     organization is indexed
     record key is f-num-agence
     access mode is random.
     select guichet assign to "fguichet.ndx"
     organization is indexed
     record key is f-num-guichet
     access mode is random.
     select compte assign to "fcompte.ndx"
     organization is indexed
     record key is f-num-compte
     access mode is random.
     select journal assign to "fjournal.seq"
     organization is sequential
     access mode is sequential.
     select ftrans assign to "transaction.seq"
     organization is sequential
     access mode is sequential.

  i-o control.
     apply lock-holding on agence, guichet, compte.

Data division.
File section.
     FD agence.
          01 art-agence.
            02 f-num-agence pic 9(3).
            02 f-solde-agence pic 9(9).
            02 filler pic x(88).
     FD guichet.
          01 art-guichet.
            02 f-num-guichet pic 9(4).
            02 f-solde-guichet pic 9(9).
            02 filler pic x(87).
     FD compte.
          01 art-compte.
            02 f-num-compte pic 9(10).
            02 f-solde-compte pic 9(9).
            02 filler pic x(81).
     FD journal.
          01 art-journal pic x(50).
     FD ftrans.
          01 art-trans.
            02 f-trans-agence pic 9(3).
            02 f-trans-guichet pic 9(4).
            02 f-trans-compte pic 9(10).
```

```
Working-storage section.

01 w-flag pic 9.
01 w-compteur pic 9(4).
01 w-timlen pic 9999 comp value 0.
01 w-d-timlen pic 9999 value 0.
01 w-timbuf pic x(24) value spaces.
01 w-return-value pic s9(9) comp value 999999999.

Procedure division.

P1 section.
Main-loop.
     Perform p2-init.
     Perform p2-trans until w-flag = 1.
     Perform p2-term.
     Stop run.

P2 section.
P2-init.
     open input ftrans.
     open extend journal allowing all.
     open i-o compte, guichet, agence allowing all.
     move 0 to w-flag.
     move 1 to w-compteur.
p2-trans.
     call "sys$asctime" using by refernece w-timlen
                              by descriptor w-timbuf
                              by value 0
                              giving w-return-value.
     display "Transaction no: " w-compteur w-timbuf.
     perform p3-lit-données.
     call "sys$asctime" using by refernece w-timlen
                              by descriptor w-timbuf
                              by value 0
                              giving w-return-value.
     display "Transaction no: " w-compteur w-timbuf.
     Perform p3-maj.
     call "sys$asctime" using by refernece w-timlen
                              by descriptor w-timbuf
                              by value 0
                              giving w-return-value.
     display "Transaction no: " w-compteur w-timbuf.
     compute w-compteur =  w-compteur + 1.

p2-term.
     close ftrans.
     close guichet.
     close agence.
     close compte.
     close journal.

p3 section.
p3-lit-données.
     read ftrans at end move 1 to w-flag.
```

```
p3-maj.
     move f-trans-agence to f-num-agence.
     read agence allowing no others
          invalid key perform p4-erreur.
     compute f-solde-agence = f-solde-agence + 5.
     rewrite art-agence allowing no others
          invalid key perform p4-erreur.
     unlock agence all records.
     move f-trans-guichet to f-num-guichet.
     read guichet allowing no others
          invalid key perform p4-erreur.
     compute f-solde-guichet = f-solde-guichet + 5.
     rewrite art-guichet allowing no others
          invalid key perform p4-erreur.
     unlock guichet all records.
     move f-trans-compte to f-num-compte.
     read compte allowing no others
          invalid key perform p4-erreur.
     compute f-solde-compte = f-solde-compte + 5.
     rewrite art-compte allowing no others
          invalid key perform p4-erreur.
     unlock compte all records.
     move "transaction-transaction-transaction"
          to art-journal.
     write art-journal.

P4 section.
p4-erreur.
     display "il y a eu une erreur dans la transaction".
```

```
Identification division.
Program-id. Bdsort.

Environment division.

Input-Output section.
File Control.
     Select fich-sortie assign to "entree"
     organization is sequential
     access mode is sequential.

Data division.

File Section.
  FD fich-sortie.
     01 art-fich-sortie.
     02 f-cle pic 9(10).
     02 f-champ pic x(90).

Working-storage section.

  01 a pic 9(5).
  01 m pic 9(10).
  01 cle1 pic 9(18).
  01 cle2 pic 9(18).
  01 quotient pic 9(18).

Procedure division.

P1 section.
 main-loop.
     perform p2-init.
     perform p2-creer-art 1000000 times.
     perform p2-term.

P2 section.
 p2-init.
     open output fich-sortie.
     move 2147467197 to m
     move 16807 to a.
     move 17557 to cle1.
 p2-term.
     close fich-sortie.
 p2-creer-art.
     perform p3-creer-cle.
     move cle1 to f-cle.
     move spaces to f-champ.
     perform p3-ecrire-art.

P3 section.
 p3-ecrire-art.
     write f-art-fich-sortie.
 p3-creer-cle.
     compute cle2 = a * cle1.
     divide m into cle2 giving quotient remainder cle1.
```

```
Identification division.
Program-id. bdjournal.

Environment division.

Input-output section.

 File-control.
     select fjournal assign to "fjournal"
     organization is sequential
     access mode is sequential.

Data division.

 File section.
   FD fjournal.
        01 art-journal pic x(50).

 Working-storage section.
    01 w-art pic x(50).

Procedure division.

P1 section.
   main-loop.
     perform p2-init.
     perform p2-create 200000 times.
     perform p2-term.
     stop run.

P2 section.
  p2-init.
     open output fjournal shared.
     move "article journal article journal article
journal    " to w-art.
  p2-term.
     close fjournal.
 p2-create.
     move w-art to art-journal.
     write art-journal.
```

```
Identification division.
Program-id. bdagence.

Environment division.

Input-output section.

 File-control.
     select fagence assign to "fagence"
     organization is indexed
     record key is f-num-agence
     access mode is dynamic.

Data division.

 File section.
   FD fagence.
        01 art-agence.
     02 f-num-agence pic 9(3).
     02 f-solde-agence pic 9(9).
     02 filler pic x(88).

 Working-storage section.
    01 w-art .
     02 w-num-agence pic 9(4).
     02 w-solde-agence pic 9(9).

Procedure division.

P1 section.
   main-loop.
     perform p2-init.
     perform p2-create 1000 times.
     perform p2-term.
     stop run.

P2 section.
  p2-init.
     open output fagence shared.
     move  0 to w-num-agence.
     move 5 to w-solde-agence.
 p2-term.
     close fagence.
 p2-create.
     move w-num-agence to f-num-agence.
     move w-solde-agence to f-solde-agence.
     write art-agence.
     compute w-num-agence = w-num-agence + 1.
```

```
Identification division.
Program-id. bdguichet.

Environment division.

Input-output section.

 File-control.
     select fguichet assign to "fguichet"
     organization is indexed
     record key is f-num-guichet
     access mode is dynamic.

Data division.

 File section.
   FD fguichet.
        01 art-guichet.
     02 f-num-guichet pic 9(4).
     02 f-solde-guichet pic 9(9).
     02 filler pic x(87).

 Working-storage section.
    01 w-art .
     02 w-num-guichet pic 9(5).
     02 w-solde-guichet pic 9(9).

Procedure division.

P1 section.
   main-loop.
     perform p2-init.
     perform p2-create 10000 times.
     perform p2-term.
     stop run.

P2 section.
  p2-init.
     open output fguichet shared.
     move  0 to w-num-guichet.
     move 5 to w-solde-guichet.
 p2-term.
     close fguichet.
 p2-create.
     move w-num-guichet to f-num-guichet.
     move w-solde-guichet to f-solde-guichet.
     write art-guichet.
     compute w-num-guichet = w-num-guichet + 1.
```

```
Identification division.
Program-id. bdcompte.

Environment division.

Input-output section.

 File-control.
     select fcompte assign to "fcompte"
     organization is indexed
     record key is f-num-compte
     access mode is dynamic.

Data division.

 File section.
   FD fcompte.
        01 art-compte.
     02 f-num-compte pic 9(10).
     02 f-solde-compte pic 9(9).
     02 filler pic x(81).

 Working-storage section.
    01 w-art .
     02 w-num-compte pic 9(10).
     02 w-solde-compte pic 9(9).

Procedure division.

P1 section.
   main-loop.
     perform p2-init.
     perform p2-create 1000000 times.
     perform p2-term.
     stop run.

P2 section.
  p2-init.
     open output fcompte shared.
     move  0 to w-num-compte.
     move 5 to w-solde-compte.
 p2-term.
     close fcompte.
 p2-create.
     move w-num-compte to f-num-compte.
     move w-solde-compte to f-solde-compte.
     write art-compte.
     compute w-num-compte = w-num-compte + 1.
```