Université
de Reims Champagne-Ardenne

INF0403

Système d'exploitation

HYPERCUBE TOKEN PIPE

Système d'exploitation

Rapport de TP

Despoullains Romain
Vautard Cloé

2023-2024

Table des matières

1 Introduction

Dans le domaine de l'informatique parallèle, la structure et l'organisation des systèmes de calcul jouent un rôle crucial dans l'efficacité et la rapidité du traitement des données. L'une des structures les plus fascinantes et complexes est celle de l'hypercube, un graphe dont les sommets peuvent représenter des processeurs interconnectés. Ce travail pratique s'inscrit dans le cadre de la Licence Informatique de l'Université de Reims Champagne-Ardenne, avec pour objectif l'implémentation d'un système de processus structurés en hypercube.

L'hypercube, ou cube n-dimensionnel, se distingue par sa capacité à illustrer efficacement les principes du calcul parallèle. Chaque sommet du cube représente un processeur, et chaque arête, une connexion directe entre deux processeurs. Cette structure permet une communication optimale entre les processeurs, facilitant ainsi le traitement parallèle des données. Dans les années 1980, les hypercubes ont révolutionné le calcul parallèle en permettant la réalisation d'ordinateurs à plusieurs processeurs, chacun traitant une partie des données simultanément.

Le présent rapport vise à décrire la mise en place d'un système de processus structurés en hypercube. Nous aborderons la théorie sous-jacente, l'architecture du projet, l'implémentation du système de communication entre processus, la gestion des signaux et des fichiers, ainsi que l'analyse des performances de notre système.

Dans le cadre de ce projet, l'organisation du code source joue un rôle crucial pour la compréhension et la maintenance du système. Les fichiers source (`.c`) sont situés dans le dossier `src`, tandis que les fichiers d'en-tête (`.h`) se trouvent dans le dossier `include`. Cette structure permet une séparation claire entre la logique d'implémentation et les déclarations d'interface.

2 Théorie et Fondements Mathématiques

2.1 Définition et propriétés d'un hypercube

Un hypercube Q_n de dimension n est un graphe dont les sommets peuvent être représentés par des chaînes de bits de longueur n. Chaque sommet est connecté à n autres sommets, formant ainsi une structure qui généralise les concepts de ligne (1D), carré (2D) et cube (3D) à des dimensions supérieures.

2.1.1 Dimension, sommets, arêtes

La dimension n d'un hypercube correspond au nombre de bits nécessaires pour représenter chaque sommet. Le nombre total de sommets dans un hypercube de dimension n est donné par la formule :

$$2^n$$

et le nombre total d'arêtes est donné par :

$$2^{n-1} \times n$$

Ces formules soulignent la croissance exponentielle du nombre de sommets et d'arêtes avec l'augmentation de la dimension de l'hypercube.

3 Architecture du Projet

L'architecture de ce projet de système de processus structurés en hypercube est conçue pour faciliter la compilation, le déploiement et la maintenance du code. La structure du répertoire est organisée comme suit :

```
.
|-- Makefile
|-- build
|   `-- hypercube
|-- clf
|   `-- clean_files.c
|-- include
|   |-- ProcessCommunication.h
|   |-- SignalHandlers.h
|   `-- Utilities.h
`-- src
    |-- HypercubeProcessSystem.c
    |-- ProcessCommunication.c
    |-- SignalHandlers.c
    `-- Utilities.c
```

3.1 Description des Composants Clés

3.1.1 Makefile et Commandes de Compilation

Le **Makefile** du projet propose plusieurs commandes pour faciliter le processus de développement, compilation, et nettoyage du système. Voici un aperçu des commandes disponibles et de leur fonction :

- `make all` ou simplement `make` : Compile tous les fichiers sources situés dans le répertoire **src**, génère les fichiers objets dans le répertoire **build** et crée l'exécutable `hypercube`. Cette commande est le point de départ pour construire l'intégralité du projet.

- `make` $(EXECUTABLE)$: Spécifiquement compile et lie les fichiers objets nécessaires pour créer l'exécutable `hypercube` sans recompiler les fichiers non modifiés. Cela optimise le temps de compilation lors du développement.

- `make build/%.o` : Compile individuellement les fichiers sources en fichiers objets. Cette commande est utile pour tester ou débugger la compilation d'un fichier spécifique sans avoir besoin de recompiler tout le projet.

— `make clean` : Nettoie le répertoire de build en supprimant tous les fichiers objets (`*.o`) et l'exécutable, ainsi qu'en exécutant les scripts de nettoyage personnalisés. Cette commande est particulièrement utile pour repartir d'une base propre avant une compilation complète ou pour s'assurer qu'il n'y a pas de fichiers résiduels pouvant causer des erreurs de compilation.

— `make clf/clean_files` : Compile et exécute le script de nettoyage `clean_files.c` situé dans le répertoire **clf**. Ce script est conçu pour effectuer des opérations de nettoyage plus spécifiques, comme supprimer des fichiers générés pendant l'exécution du programme.

— `make test` : Compile le projet et exécute l'exécutable `hypercube` pour tester rapidement les fonctionnalités du système. Cette commande peut être adaptée pour lancer des tests automatisés spécifiques au projet.

Ces commandes fournissent un ensemble d'outils complet pour gérer le cycle de vie du développement du système de processus structurés en hypercube, de la compilation à la maintenance en passant par le test.

3.1.2 Code Source

Les fichiers source situés dans le répertoire **src** implémentent la logique principale du système d'hypercube. Chaque fichier .c correspond à une partie spécifique de la fonctionnalité du système :

— `HypercubeProcessSystem.c` initialise et gère l'hypercube.

— `ProcessCommunication.c` gère la communication entre les processus.

— `SignalHandlers.c` implémente la gestion des signaux pour contrôler les processus.

— `Utilities.c` contient des fonctions utilitaires.

4 Implémentation

L'implémentation du système d'hypercube est orchestrée principalement à partir du fichier `HypercubeProcessSystem.c`, où se trouve la fonction `main`. Ce fichier coordonne l'initialisation du système et gère son exécution. Dans `ProcessCommunication.c`, nous utilisons deux constantes de préprocesseur, `ENABLE_DISPLAY_DESCRIPTION` et `ENABLE_DISPLAY_TOKEN_JOURNEY`, qui, lorsqu'activées, fournissent des informations détaillées sur les tubes de l'hypercube et le parcours du jeton, respectivement, facilitant ainsi le débogage et la compréhension du système en temps réel.

4.1 HypercubeProcessSystem.c

Ce fichier est responsable de l'initialisation et de la gestion de l'hypercube. La fonction principale `main` initialise le système en suivant ces étapes :

1. Vérification des arguments passés au programme pour s'assurer que la dimension de l'hypercube est fournie.

2. Conversion de la dimension en entier et initialisation des structures de données nécessaires.

3. Appel de `delete_previous_files` pour nettoyer l'environnement avant l'exécution.

4. Appel de `init_pipes` pour initialiser les tubes de communication entre les processus.

5. Création des processus représentant les sommets de l'hypercube via `create_hypercube_processes`.

6. Attente de la terminaison de tous les processus enfants avec `wait`

4.2 ProcessCommunication.c

Ce fichier implémente les mécanismes de communication entre les processus. Les fonctions clés incluent :

— `init_pipes(int n)` : Initialise `nb_pipes` tubes pour la communication. Le nombre de tubes est calculé comme $2^n \times n$, où n est la dimension de l'hypercube.

— `create_hypercube_processes(int n)` : Crée 2^n processus, chacun représentant un sommet de l'hypercube. Chaque processus est connecté à ses voisins via les tubes appropriés.

— `token_journey(int id_process, int *pipe_ids_list, int n)` : Gère la circulation d'un jeton à travers l'hypercube, illustrant un parcours aléatoire.

4.3 SignalHandlers.c

Ce fichier est dédié à la gestion des signaux pour suspendre et reprendre l'exécution des processus. Les fonctions importantes sont :

— `father_handler(int sig)` : Traite les signaux reçus par le processus père pour suspendre ou reprendre les processus enfants.

— `child_sigint_handler(int sig)` : Permet aux processus enfants de terminer proprement en réponse à un signal d'interruption.

4.4 Utilities.c

Ce fichier contient des fonctions utilitaires, notamment :

— `delete_previous_files()` : Supprime les fichiers créés lors des exécutions précédentes pour éviter les conflits.

— `isInTab(int val, int* tab, int size)` : Vérifie si une valeur est présente dans un tableau. Cette fonction est utilisée pour gérer les descripteurs de fichier ouverts.

4.5 Exemples de Code et Explications

4.5.1 Initialisation des Tubes

La fonction `init_pipes` utilise un calcul mathématique pour déterminer le nombre de tubes nécessaires, basé sur la dimension de l'hypercube :

```
nb_pipes = (1 << n) * n;
```

Cette ligne calcule $2^n \times n$, le nombre total de tubes nécessaires pour un hypercube de dimension n.

4.5.2 Communication entre Processus

Dans `create_hypercube_processes`, chaque processus enfant est connecté à ses voisins. La connexion utilise l'opération XOR pour déterminer les voisins :

```
int other_p = id_process ^ (1 << i);
```

Cette opération trouve un voisin en changeant un seul bit dans l'identifiant du processus, illustrant la propriété d'adjacence dans l'hypercube.

5 Mise en Place de la Structure de Communication

La communication entre les processus dans notre système d'hypercube est établie via des tubes (pipes) UNIX, permettant une communication bidirectionnelle entre sommets adjacents. Chaque sommet du hypercube est représenté par un processus, et chaque arête du hypercube par un couple de tubes pour la communication bidirectionnelle.

La dimension de l'hypercube n'étant pas connue lors de la compilation, il est nécessaire d'allouer dynamiquement de la mémoire pour les différents tableaux. La fonction atexit(...) est utilisée après chaque allocation dynamique de mémoire pour permettre à la fonction exit(...) d'appeler une fonction qui liberera la mémoire allouée avant de terminer le processus.

Chaque processus dans notre système d'hypercube communique avec ses voisins en utilisant un modèle précis de tubes pour la lecture et l'écriture. Par exemple, un processus d'identifiant `id_process` communique avec un processus d'identifiant `other_p` en lisant dans le tube `pipe_fds[id_process * n + i][0]` pour la réception des données et en écrivant dans `pipe_fds[other_p * n + i][1]` pour l'envoi des données, où `i` correspond à la position du bit différent. Pour chaque processus, les descripteurs de fichier de lecture sont stockés aux indices 2*i dans le tableau pipe_ids_list et ceux pour l'écriture sont stockés aux indices 2*i+1. Cette méthode assure une communication efficace et structurée entre les processus adjacents.

5.1 Création et Gestion des Tubes

La fonction `init_pipes(int n)` initialise un ensemble de tubes nécessaires pour la communication entre tous les sommets adjacents d'un hypercube de dimension n. Le nombre total de tubes créés est déterminé par la formule $2^n \times n$, où 2^n est le nombre de sommets et n est le nombre de sommets adjacents à chaque sommet.

```
for (int i = 0; i < nb_pipes; i++) {
    pipe_fds[i] = (int *) malloc(2 * sizeof(int));
    if (pipe(pipe_fds[i]) == -1) {
        perror("Pipe initialization failed");
        exit(EXIT_FAILURE);
    }
}
```

Cette boucle crée les tubes nécessaires et stocke leurs descripteurs de fichier dans un tableau pour une utilisation ultérieure. Chaque tube est bidirectionnel, permettant la communication dans les deux sens entre deux processus adjacents.

6 Parcours du Jeton et Gestion des Signaux

Le parcours du jeton est géré à travers une combinaison de la fonction `set_readfds` pour la préparation des ensembles de lecture et l'utilisation de `select` pour attendre activement la réception du jeton. Cela permet à chaque processus de réagir rapidement à l'arrivée du jeton et de déterminer le moment précis pour lire les données. La gestion des signaux est cruciale pour le contrôle du flux de travail, où `father_handler` et `child_sigint_handler` jouent des rôles centraux dans la suspension, la reprise et la terminaison propre des processus en réponse aux signaux `SIGUSR1` et `SIGINT`.

6.1 Parcours du Jeton

Le parcours du jeton à travers l'hypercube est implémenté de manière aléatoire. Au démarrage, le processus représentant le sommet initial (généralement 00...0) génère un jeton et l'envoie à l'un de ses voisins choisis aléatoirement. Chaque processus recevant le jeton choisit ensuite, de manière aléatoire, l'un de ses voisins à qui envoyer le jeton.

La sélection aléatoire du voisin est réalisée en choisissant aléatoirement un bit à inverser dans l'identifiant binaire du processus courant, déterminant ainsi l'identifiant du processus voisin dans le hypercube.

```
position_bit_dif = rand() % n;
if (write(pipe_ids_list[2*position_bit_dif+1], &token, sizeof(int)) == -1) {
    perror("Write to pipe failed");
    exit(EXIT_FAILURE);
}
```

Cette portion de code sélectionne un tube de sortie aléatoire parmi ceux connectés au processus courant et y écrit le jeton.

6.2 Gestion des Signaux

La gestion des signaux est utilisée pour suspendre et reprendre l'exécution des processus, ainsi que pour terminer proprement le système. Le processus père utilise le signal `SIGUSR1` pour alterner entre la suspension et la reprise de l'exécution des processus enfants. La réception de `SIGINT` provoque la terminaison du système.

```
void father_handler(int sig) {
    switch(sig) {
        case SIGUSR1:
            suspend_flag = !suspend_flag;
            for(int i = 0; i < nb_processes; i++)
                kill(child_pids[i], suspend_flag ? SIGSTOP : SIGCONT);
            break;
        case SIGINT:
            for(int i = 0; i < nb_processes; i++)
                kill(child_pids[i], SIGINT);
            break;
    }
}
```

Ce gestionnaire de signaux alterne l'état de suspension des processus enfants et assure leur terminaison propre en réponse à un signal d'interruption.

7 Enregistrement et Analyse des Visites

L'une des fonctionnalités clés de notre système d'hypercube est l'enregistrement et l'analyse des visites du jeton par les différents processus. Cette section décrit le mécanisme d'enregistrement des données et leur analyse postérieure pour comprendre le comportement du système.

Le système prend des mesures proactives pour assurer un environnement de travail propre en supprimant tous les fichiers de log précédents au démarrage. Chaque processus fils enregistre des informations vitales sur le jeton dans un fichier dédié, permettant une analyse approfondie post-exécution du comportement du système et du parcours du jeton. Cette méthode d'enregistrement fournit des insights précieux sur la performance et l'efficacité de la communication au sein de l'hypercube.

7.1 Suppression des Fichiers Précédents

Au démarrage du programme, une opération de nettoyage est effectuée pour supprimer tous les fichiers de nom correspondant au modèle `[nombre].txt` dans le répertoire courant. Cette étape

assure que l'exécution du programme démarre dans un environnement propre, sans interférence des données résiduelles d'exécutions précédentes.

```
void delete_previous_files() {
    DIR *dir;
    struct dirent *entry;
    dir = opendir(".");
    if (dir == NULL) {
        perror("Failed to open directory");
        exit(EXIT_FAILURE);
    }
    while ((entry = readdir(dir)) != NULL) {
        if (isLogFile(entry->d_name)) {
            unlink(entry->d_name);
        }
    }
    closedir(dir);
}
```

7.2 Enregistrement des Données

Chaque processus fils, après avoir été créé et avoir établi sa communication via les tubes, ouvre (ou crée s'il n'existe pas) un fichier nommé `"[id_process].txt"`. C'est dans ce fichier que les données relatives à la réception du jeton sont enregistrées.

Pour chaque visite du jeton, le processus écrit dans son fichier associé deux informations principales : la valeur courante du jeton et le temps écoulé depuis sa dernière réception, mesuré en secondes. Cette opération permet de tracer le chemin du jeton à travers l'hypercube et d'analyser la fréquence des visites pour chaque sommet (processus).

7.3 Analyse des Visites

L'analyse des fichiers de log générés par chaque processus offre une vision précise de la distribution du jeton à travers l'hypercube. En examinant les intervalles de temps entre les réceptions successives du jeton, nous pouvons déduire des modèles de circulation du jeton et identifier d'éventuels goulots d'étranglement ou inefficacités dans la structure de communication.

Cette analyse peut être réalisée à l'aide de scripts ou de programmes supplémentaires qui parcourent les fichiers de log, synthétisent les données et génèrent des rapports ou des visualisations illustrant le comportement du système durant son exécution.

8 Conclusion

Ce projet a permis de concrétiser la théorie des hypercubes dans une application pratique de calcul parallèle, illustrant la puissance et l'efficacité de la communication dans une structure d'hy-

percube. En naviguant à travers les défis de la mise en place d'un système de communication efficace, la gestion du parcours aléatoire du jeton, et la manipulation adéquate des signaux, nous avons approfondi notre compréhension des systèmes parallèles et de la synchronisation des processus.

L'implémentation réussie de ce projet révèle non seulement la viabilité de l'hypercube comme modèle pour le calcul parallèle, mais aussi l'importance d'une conception soignée pour optimiser la communication et le traitement parallèles. La capacité de visualiser le parcours du jeton et d'analyser les interactions entre processus a été essentielle pour déboguer et affiner le système, offrant une plateforme solide pour l'expérimentation et l'apprentissage.

Bien que le système mis en place fonctionne conformément aux attentes, des améliorations pourraient être envisagées pour augmenter la robustesse, l'efficacité et la scalabilité du système. Explorer d'autres variantes d'hypercubes ou intégrer des algorithmes de parcours plus complexes pourrait offrir de nouvelles perspectives sur le calcul parallèle et sur les manières d'optimiser la communication inter-processus.

En conclusion, ce projet a non seulement renforcé notre compétence technique dans la réalisation de systèmes parallèles complexes, mais a également élargi notre vision sur les possibilités et les défis du calcul parallèle. L'avenir de l'informatique parallèle s'annonce riche et prometteur, et les leçons apprises au cours de ce projet fourniront une base solide pour nos futures explorations dans ce domaine fascinant.

Merci de votre lecture !