Travaux d'Etudes et de Recherches

Simulation coopérative et parallèle : Expérimentations avec le scheduler SystemC

Préparé par :

Mohamed Zaim Wadghiri
Mohamed.Zaim-Wadghiri@imag.fr

Encadré par :

Matthieu Moy
Matthieu.Moy@imag.fr

Claire Maiza
Claire.Maiza@imag.fr

Février - Mai 2011

Simulation coopérative et parallèle : Expérimentations avec le scheduler SystemC

Mohamed Zaim Wadghiri

Grenoble INP - Ensimag

Mai 2011

Résumé

SystemC est un langage de description de matériel (HDL : Hardware Description Language) permettant une modélisation de haut niveau des systèmes sur puce (SoC), que ce soit au niveau matériel ou logiciel. C'est une bibliothèque C++ qui modélise chaque bloc matériel du SoC par une classe appelée module, et décrit son comportement à l'aide de processus.

La simulation dans SystemC est gérée par un scheduler qui n'est pas préemptif. La simulation est donc dite coopérative et n'exploite qu'un seul processeur sur la machine. L'objet du travail réalisé dans ce TER est d'implémenter une parallélisation de SystemC pour pouvoir bénéficier de la puissance des machines actuelles et d'exécuter la simulation sur les différents processeurs. Ceci va permettre alors de gagner considérablement en temps de simulation.

Mots-clés : SystemC, Simulation coopérative, Simulation parallèle, Threads POSIX

Tuteurs : Matthieu Moy et Claire Maiza

Table des matières

Table des figures

Liste des algorithmes

Remerciements

Je tiens à remercier énormément mes deux encadrants Matthieu Moy et Claire Maiza pour l'aide qu'ils m'ont toujours apportée durant ces 3 mois et pour le temps qu'ils m'ont consacré, ainsi qu'à toute l'équipe Synchrone qui m'a accueillie et plus particulièrement à sa directrice Florence Maraninchi, qui a eu la bonne idée de créer ce module dans le parcours proposé à l'Ensimag.

Première partie
Introduction

1 Contexte du travail

Les travaux présentés dans ce document s'inscrivent dans le cadre du module Travail et Etudes de Recherche. Le projet TER est un module qui fait partie de notre formation à l'Ensimag, et dont le but est de découvrir le monde de la recherche ainsi que de travailler dans un laboratoire grenoblois sur un sujet proposé par des enseignants-chercheurs.

D'une part, le TER permet de découvrir un sujet technique un petit peu avancé, et qui reste en parfaite harmonie avec notre parcours à l'Ensimag. D'autre part, il reste une occasion d'apprendre des méthodes avec lesquelles on travaille dans une équipe de recherche.

Ainsi, nous allons présenter tout au long de ce document une synthèse du travail réalisé durant ces 3 mois au sein du laboratoire Verimag, et qui portait sur la parallélisation de SystemC.

2 Présentation de l'équipe de recherche

Ce projet TER a été encadré par Matthieu Moy et Claire Maiza, des enseignants-chercheurs au sein de l'équipe Synchrone du laboratoire Verimag.

Verimag est un laboratoire leader de la recherche dans le domaine des systèmes embarqués, que ce soit du côté logiciel ou matériel. De plus, il dispose d'un large partenariat avec plusieurs entreprises industrielles pour travailler ensemble sur des sujets de pointe, notamment STMicroelectronics.

Le laboratoire est organisé en 3 grandes équipes : Synchrone, DCS et Tempo.

Personnellement, j'étais accueilli par l'équipe Synchrone qui est spécialisée dans le domaine des langages synchrones et des systèmes réactifs. Elle est très connue pour son développement du langage Lustre qui a commencé au début des années 1980, et dont la version commerciale est Scade, largement répandue dans plusieurs entreprises telles que Airbus, Dassault, Schneider Electric ...

3 Présentation du problème

SystemC est un langage de description de matériel (HDL : Hardware Description Language) tout comme VHDL et Verilog, mais qui permet aussi une modélisation de plus haut niveau des systèmes sur puce, que ce soit au niveau matériel ou logiciel. Chaque bloc est alors modélisé par une classe C++ appelée module où on décrit son comportement à l'aide de processus.

Le vrai système à concevoir dispose de différents blocs qui vont fonctionner de façon purement parallèle, ainsi la simulation devrait aussi respecter cette contrainte. Or, les différents processus définis dans les modules sont gérés par le scheduler SystemC qui, lui, n'est pas

préemptif. Ainsi, on a une simulation coopérative du circuit à implanter au lieu d'une simulation qui devrait être parallèle. Ceci a alors un effet sur le temps de simulation qui devient vite très grand surtout si on traite des circuits très complexes, et n'exploite pas le parallélisme sans cesse croissant de nos machines actuelles.

Ainsi, le but de ce TER est de proposer une implémentation d'une méthode pour mélanger la simulation parallèle (avec threads POSIX) et la simulation coopérative des processus SystemC. Cette API est définie au-dessus de SystemC : les programmes déjà écrits doivent toujours fonctionner.

Deuxième partie

Modélisation transactionnelle des systèmes sur puce

Depuis quelques années déjà, des appareils électroniques “intelligents” sont devenus omniprésents dans notre vie quotidienne. De plus, ceci ne cesse de s’accroître de jour en jour. Assis devant votre PC, vous avez certainement à côté de vous un téléphone portable, une carte SIM, une télévision, un récepteur satellite ... Dans votre portefeuille, une carte vitale, une carte bleue, ... En prenant la voiture, un système GPS, un “bouton” pour l’ouverture automatique des différentes vitres ... Pourtant, si on prend l’exemple d’une carte bancaire, on constate que c’est un minuscule objet qui réalise des tâches complexes !

Tous ces systèmes ont été intégrés dans un petit circuit assemblant tous les composants nécessaires à leur fonctionnement (un processeur ou plusieurs, de la mémoire, des interfaces Entrée / Sortie, d’autres composants plus spécifiques ...).

Ce système est appelé un système sur puce (en anglais, System-on-Chip : SoC).

1 Flot de conception (Design Flow)

1.1 Présentation générale et contraintes du Design Flow

La conception des systèmes sur puce et des circuits intégrés de façon générale passe naturellement par plusieurs étapes et divers niveaux d’abstraction. Ce processus est alors appelé un flot de conception. Ce dernier dépend aussi de la complexité du système et du nombre de fonctionnalités qu’il doit réaliser. Cette complexité ne cesse d’augmenter pour respecter différentes contraintes :

- Taille minuscule de la puce
- Puissance de calcul croissante
- Consommation énergétique garantissant une bonne autonomie du système
- Time To Market (TTM) : il faut être parmi les premiers dans le marché
- Time To Money (TTM aussi !) : il faut reprendre les bénéfices dans les plus brefs délais

Ceci reste alors une tâche délicate. En effet, il est nécessaire de faire des simulations du SoC avant même de commencer sa production (Prototypage Virtuel) pour deux raisons :

- Tester et valider les systèmes pour éviter la fabrication d’un produit défectueux
- Essayer de convaincre les clients (qui sont dans la plupart des cas, des entreprises de poids) que le produit sera le meilleur en leur fournissant des prototypes de simulation.

1.2 Le niveau RTL et le problème du “Design Gap”

Le niveau RTL (Register Transfer Level) est un niveau d’abstraction du système utile et efficace. En effet, la modélisation du système se base dans ce modèle, sur l’interconnexion de registres qui échangent des signaux. Ce niveau d’abstraction est utilisé par VHDL et Verilog.

Son avantage, c'est qu'il permet une synthèse en "netlist" (description au niveau portes) assez facile. Après un placement routage puis une impression sur silicim, on aboutit au circuit électrique proprement parlé.

Le problème qui se pose provient du fait que la productivité des designers croît d'environ 30% par an. Pourtant, au niveau matériel, la loi de Moore [1] convient que la complexité des systèmes ne fait qu'augmenter et devenir encore plus complexe deux fois plus chaque 18 mois. Ainsi, la simulation par VHDL par exemple, devient vite très dure pour les designers puisque la quantité de code à produire et à corriger devient grande. Cet écart est appelé "Design Gap"[1]. Nous tenons à mentionner que plusieurs travaux ont été réalisés à propos du "Bridging the Design Gap".

1.3 Vers une modélisation transactionnelle : le modèle TLM

Le modèle TLM (Transaction Level Modeling) est un modèle d'abstraction de haut niveau, où la structure et l'interconnexion des registres est cachée dans des modules. On s'interesse alors davantage au côté fonctionnel en décrivant ces modules sous forme de processus intéragissant de façon concurrente. Ce niveau est utile tant sur le plan de simulation que celui de la validation.

Le modèle TLM est basé sur des échanges entre les différents modules du système, qu'on appelle aussi "transaction", d'où l'appelation de ce niveau d'abstraction. Ceci permet de gagner en vitesse de simulation mais aussi de réduire les lignes de codes que le développeur est amené à écrire.

2 SystemC

SystemC est un langage de description de matériel permettant une modélisation transactionnelle des systèmes sur puce, du niveau matériel au logiciel. Plus précisément, SystemC n'est pas un langage, mais plutôt une librairie à part entière de C++, ainsi il profite de toute la puissance de la programmation objet.

SystemC est une librairie qui fournit toutes les constructions nécessaires pour créer efficacement un modèle d'algorithmes, d'architecture matérielle mais aussi des interfaces des SoC. Ainsi, on peut à la fois simuler, valider et optimiser le système.

2.1 Structure d'un programme SystemC

SystemC ajoute une API à C++ permettant la modélisation des systèmes suivant une structure qui concordre avec les principes du modèle TLM. Ce qui rend SystemC plus intéressant, c'est qu'il permet ce qu'on appelle le hardware-software co-design [2].

La structure d'un programme SystemC est basée sur un ensemble d'éléments. On présente dans ce rapport seulement les deux éléments utilisés le plus fréquemment dans ce TER :

1. Gordon Moore, cofondateur de la société Intel, avait affirmé en 1965 que "le nombre de transistors par cicuit de même taille va doubler tous les 18 mois"

2. Le Hardware-Software codesign peut être défini comme la conception simultanée du matériel et des logiciels à mettre en œuvre. Ceci va de même pour la co-vérification, qui est la vérification simultanée du logiciel et du matériel pour valider les fonctionnalités désirées dans le système.

▶ **Module :** C'est le composant le plus haut de la hiérarchie. C'est un conteneur qui représente un grand bloc du système, qui lui même peut contenir d'autres modules et des processus.

▶ **Process :** Il permet de décrire le fonctionnement du système. Il existe 3 types :

- `SC_METHOD` : sont équivalents à de simples méthodes C++. Ils sont appelés par le scheduler SystemC à chaque notification d'événement appartenant à leur liste de sensibilité [3], et s'exécutent entièrement pour finir avec `return` qui redonne la main au scheduler.
- `SC_THREAD` : sont des threads SystemC, qui sont lancés au début de la simulation et qui sont des boucles infinies qui doivent être interrompues pour donner la main aux autres threads.
- `SC_CTHREAD` : représentent un cas particulier de `SC_THREAD` qui, de plus, sont synchrones (Clocked Thread).

La figure 1 représente une illustration de l'organisation de SystemC tirée du manuel de référence de SystemC. [4]

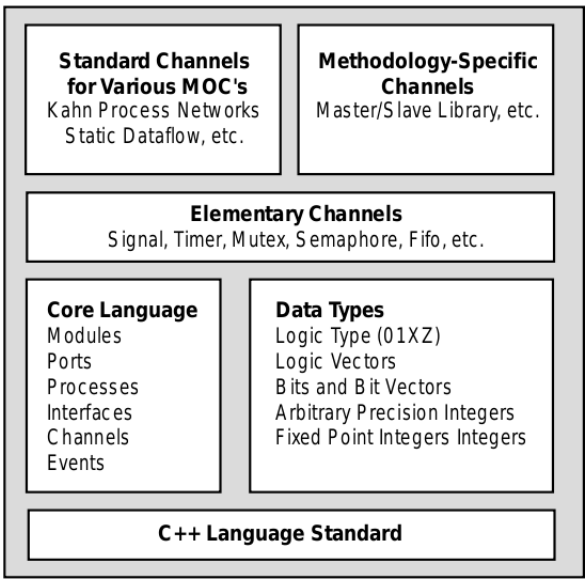


FIGURE 1 – Architecture et organisation de SystemC

3. La liste de sensibilité est une liste de signaux qui déclenchent l'activité du processus à chaque changement de valeur de l'un d'eux.

2.2 Le scheduler SystemC : Définition du problème

2.2.1 Temps simulé et "Wall-Clock Time"

Il faut distinguer dès maintenant deux notions du temps différentes : le temps simulé et le "Wall-Clock time".

Le temps simulé est le temps que prend un processus ou une tâche pour s'exécuter dans une implémentation matérielle.

Le "Wall-Clock time" est le temps qu'un développeur attend pour que le processus de simulation du système termine. Par conséquent, il est toujours souhaitable d'avoir un temps de simulation qui est rapide.

2.2.2 L'ordonnanceur SystemC

SystemC dispose d'un ordonnanceur qui permet de gérer les différents processus lancés. Tout d'abord, rappelons le graphe des états de processus (Fig. 2). Un processus peut être dans l'un des 3 états suivants :

- Elu : s'exécute et utilise le processeur à l'instant courant
- Eligible : prêt à être élu et attend que le scheduler le choisisse
- Bloqué : attend une donnée (E/S, événement, ...)

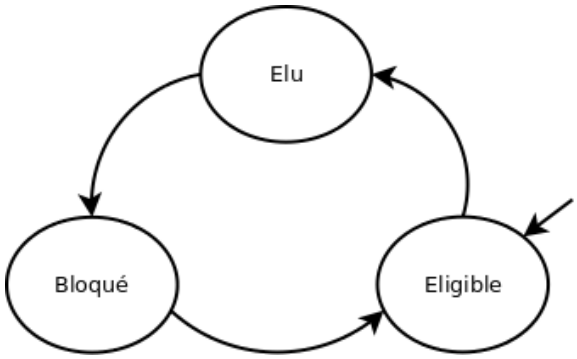


FIGURE 2 – Etats d'un processus

L'ordonnanceur SystemC passe par plusieurs étapes pour sélectionner les processus qu'il doit exécuter. Ceci est organisé en 4 grandes phases [5] :

1. **Initialisation :** exécution de tous les processus. A noter que le choix de l'ordre d'exécution des différents processus est indéterministe.

2. **Evaluation :** durant cette phase, les processus s'exécutent et se suspendent eux-même par des instructions `wait` (attendre une durée ou un événement). Quand il n'y a plus de processus éligibles, on passe à l'étape suivante.

3. **Update :** le noyau SystemC effectue alors une mise à jour des différents signaux. Les étapes 2 et 3 représentent ce qu'on appelle un delta-cycle.

4. **Time Elapse :** ce n'est qu'à cette étape qu'on fait avancer le temps simulé.

La figure 3 est une synthèse du mécanisme de fonctionnement du scheduler SystemC :

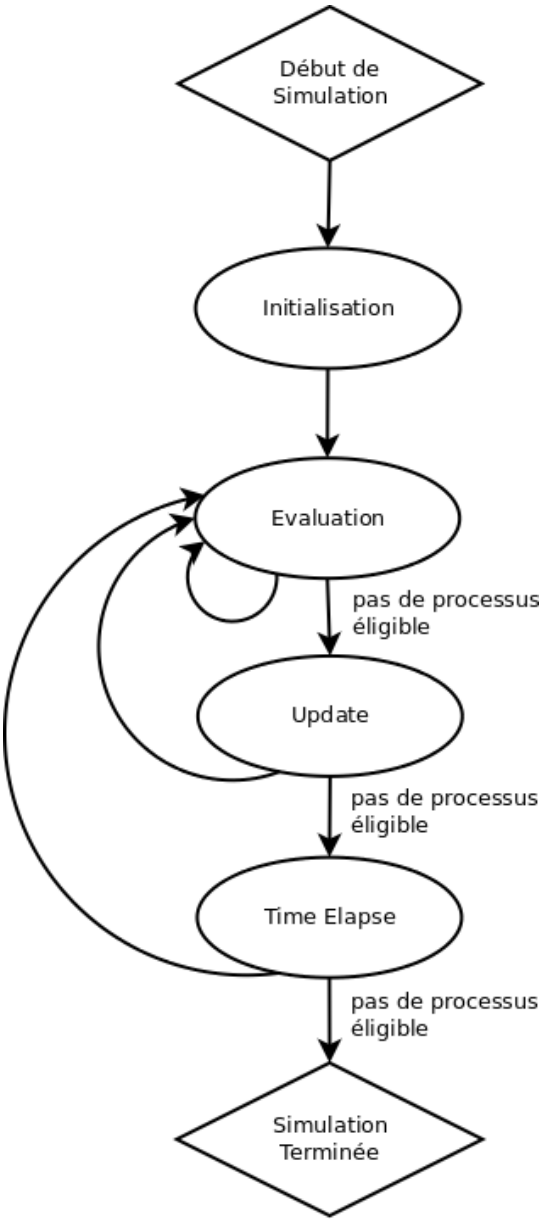


FIGURE 3 – Mécanisme de fonctionnement du scheduler SystemC

2.2.3 Simulation coopérative

Le scheduler SystemC n'est pas préemptif. Ainsi, l'exécution des différents processus SystemC se fait de façon coopérative. Le scheduler ne fait alors que donner la main à un autre processus et ne l'interrompt pas lors de son exécution. Les programmeurs SystemC doivent alors introduire dans le code du processus des points de synchronisation pour pouvoir rendre la main à un autre processus, sinon on aura une boucle infinie non interruptible.

Pour résumer, **le scheduler SystemC est multitâche coopératif : un seul processus est exécuté à un instant donné qui, de plus, n'est pas préemptible**).

2.2.4 Simulation parallèle

Depuis quelques années déjà, les ordinateurs ainsi que les super-calculateurs disposent de plus en plus de pocesseurs multi-coeurs. La simulation d'un modèle écrit en SystemC ne bénéficie pas du tout de la vitesse de ces puissantes machines, et le temps de simulation qui prend des fois plusieurs semaines ne peut être réduit ! L'idée alors est de pouvoir rendre l'exécution des threads SystemC parallèle.

Dans le cadre de ce TER, on va plutôt s'intéresser à une parallélisation à sémantique constante, c'est-à-dire, les programes déjà écrits en SystemC doivent rester fonctionnels en ajoutant l'API que l'on va implémenter. Ceci est dû, tout d'abord, à la complexité de la bibliothèque SystemC, et donc, un TER ne laisse pas le temps nécessaire pour pouvoir la modifier, mais aussi, on aimerait bien que les programmes déjà fonctionnels le restent.

Il est à noter qu'un travail en parallèle avec celui-ci, et qui traite du même sujet mais en modifiant l'algorithme du scheduler SystemC, est en cours de réalisation par Samuel Jones dans le cadre d'un stage de Master Recherche.

Troisième partie

Parallélisation de SystemC par des threads POSIX

1 La bibliothèque pThread

Un thread (ou processus léger) est un fil d'exécution semblable à un processus normal sauf que les différents threads d'un processus partagent la mémoire virtuelle au moment où chaque processus a la sienne. Ainsi, si on gagne de la rapidité et du parallélisme, il faut se méfier des accès concurrents à des variables partagées qui peut entraîner des valeurs incohérentes. Heureusement, il existe plusieurs mécanismes pour réaliser des exclusions mutuelles dont les mutex, les sémaphores, les moniteurs ... [2]

La norme POSIX fournit une bibliothèque `pThread` (Posix Thread) de fonctions qui permettent de créer et manipuler les threads.

2 Notion de "tâche avec durée"

L'idée sur laquelle on s'est basée pour implémenter la parallélisation de SystemC est empruntée au principe de durée avec tâche de jTLM[3]. jTLM est un simulateur préemptif écrit en Java qui a été développé à Verimag.

Comme on l'a précisé à la section 5.2.2, le scheduler SystemC ne fait avancer le temps de simulation qu'à la phase "Time Elapse". L'exécution du processus pendant un delta-cycle se fait alors en un temps nul du point de vue de SystemC.

Le principe de base de notre solution c'est qu'un `SC_THREAD` pourra déléguer une partie de sa fonction à un `pThread` s'exécutant pendant une durée donnée. Cette durée est passée à un `wait` pour attendre l'exécution de la tâche. On bénéficie alors du parallélisme de l'OS dont le scheduler sélectionne les différents `pThreads` à s'exécuter de façon parallèle.

3 Première approche : One-To-One `SC_THREAD - pThread`

La solution la plus naïve à laquelle on pourrait penser au tout début, c'est d'associer à chaque processus SystemC un pThread. On bénéficie dans ce cas de la préemptivité du scheduler du système d'exploitation qui donne la main à chaque fois à un processus donné (Fig. 4).

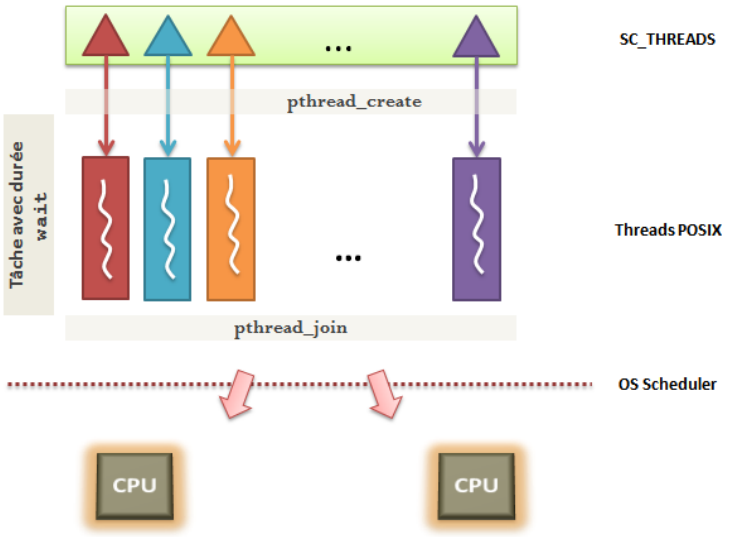


FIGURE 4 – Principe de la solution One-To-One `SC_THREAD - Thread POSIX`

Imaginons que nous disposons de 3 `SC_THREAD` s'exécutant en SystemC (on les notera A, B et C). Pour pouvoir les exécuter en parallèle, il faut suivre les étapes suivantes :

1. Mettre le corps de chaque processus dans une fonction. On aura alors 3 fonctions `fct_A`, `fct_B` et `fct_C`.
2. Créer pour chaque `SC_THREAD` un thread Posix dont la routine est la fonction correspondante.
3. Attendre un certain temps grâce à la fonction `wait` qui donne au processus une notion de durée.

Dans la libraire `during.h`, on dispose alors d'une fonction `void during_threads(double time_ns, sc_time_unit unit, void *(*routine) (void *), sc_core::sc_module *m)`, à laquelle il faut passer la durée de la tâche et son unité ainsi que la routine à exécuter.

Algorithm 1 Algorithme d'implémentation de la solution One-To-One `SC_THREAD - Thread POSIX`

```
1 void during_threads(double time_ns, sc_time_unit unit,
2                     void *(*routine) (void *), sc_core::sc_module *m) {
3     pthread_create(&thr, NULL, routine, m);
4     wait(time_ns, unit);
5     pthread_join(thr, &st);
6 }
```

En utilisant cette approche, le programmeur n'a qu'à implémenter le comportement d'un bloc dans une fonction `fct`. Ensuite, il doit appeler la fonction `during_threads` dans le `SC_THREAD` avec, pour paramètres, `fct` et une durée d'exécution.

A la fin, on disposera d'un nombre de `pThread` égal à celui des différents `SC_THREADS` de tous les modules s'exécutant de façon parallèle.

La création d'un `pThread` consite à des appels systèmes qui nécessitent des changements de contextes et donc, qui prennent un temps non négligeable surtout dans le cas où le `SC_THREAD` ne réalise pas une tâche assez importante (un programme vide par exemple). Ainsi, on pourra avoir une solution parallèle beaucoup plus lente que la version séquentielle elle-même ! Ceci se voit dans les tests que nous avons effectués et qui seront présentés dans la partie 4. Nous avons alors pensé à une solution moins naïve et qui pourra remédier à ce problème. C'est le sujet de la section suivante.

4 Deuxième approche : Parallélisation avec un modèle Producteur / Consommateur

Dans cette version, on limite le nombre de `pThreads` qui sont créés en le définissant à l'avance et en les créant une fois pour toute (Alg. 3). En fait, on n'associera plus un `pthread` à chaque `SC_THREAD`, mais l'utilité de ces `pThread` sera un rôle de lecteur dans un modèle producteur/consommateur classique [2] (Fig. 5).

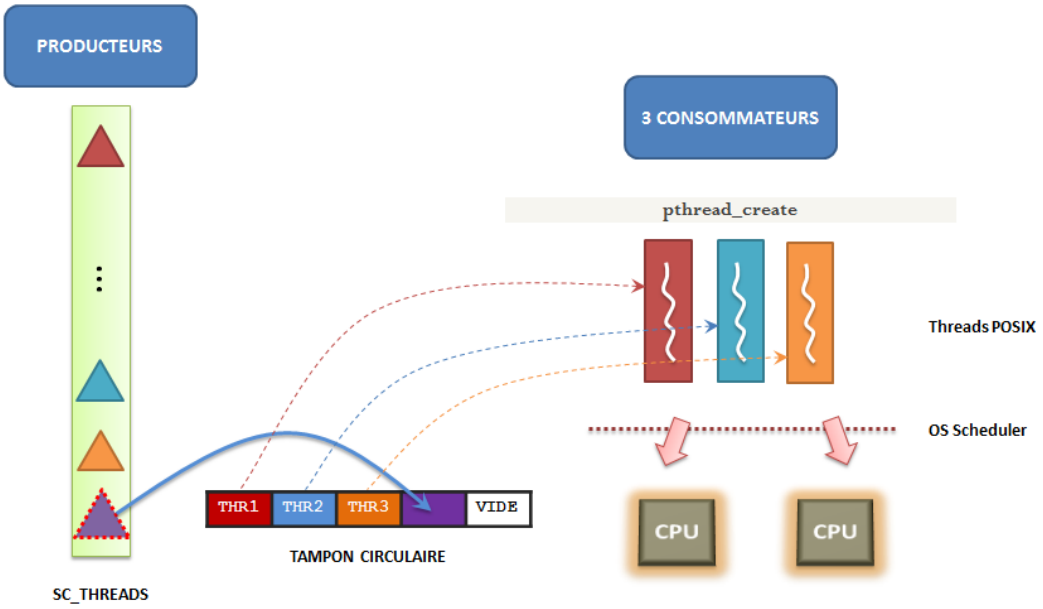


FIGURE 5 – Principe de la solution basée sur le modèle producteur/consommateur (Cas de 2 processeurs)

En effet, on distinguera deux "milieux" différents : SystemC et threads POSIX au niveau de l'OS. La communication entre les deux milieux se fait grâce à un tampon (circulaire). Les `SC_THREADS` déposent le code à exécuter en parallèle sur le tampon (Alg. 2), et les `pThreads` consommateurs récupérent les fonctions à exécuter.

Algorithm 2 Implémentation de l'approche producteur/consommateur

```
1 void during_prod_cons(double time, void *(*routine) (void *)) {
2     ecrire_tampon(routine);
3     wait(time, SC_NS);
4 }
```

Du côté pthreads, l'instanciation des différents pthreads lecteurs se fait dans la fonction `init_prod_cons`. Elle doit être appelée avant le début de simulation (avant `sc_start`).

Algorithm 3 Initialisation des sémaphores et du tampon dans le modèle producteur/consommateur

```
1 void init_prod_cons() {
2     //initialisation du tampon et des différents sémaphores
3     [...]
4     for (int i = 0; i < 3; i++)
5         pthread_create(&tabpthr[i], NULL, lire_tampon, NULL);
6 }
```

La fonction de lecture permet en effet de lire le tampon s'il n'est pas vide, puis de l'exécuter.

Le choix du nombre de pthreads lecteurs peut dépendre par exemple du nombre de processeurs de la machine sur laquelle on exécute la simulation en SystemC. En effet, le choix de `NB_THREADS = NB_PROC+1` où `NB_PROC` est le nombre de processeurs, semble assez convaincant puisqu'on aura un pthread par processeur, ainsi qu'un autre de plus pour prendre le relai si l'un des autres est en attente d'une entrée/sortie par exemple.

Pour résumer, supposons qu'on dispose de 3 `SC_THREADS` A, B et C. L'utilisation de cette solution passe alors par les étapes suivantes :

1. Associer à chaque `SC_THREAD` une fonction implémentant son corps. On aura respectivement 3 fonctions `fct_A, fct_B` et `fct_C`.
2. Appeler la fonction `during_prod_cons` dans chaque `SC_THREAD` avec la durée qu'on veut attribuer à la tâche ainsi que la fonction correspondante.
3. Initialiser le modèle (tampon et sémaphores) grâce à la fonction `init_prod_cons` avant le début de simulation (avant `sc_start`).

Quatrième partie

Tests et évaluations des différentes approches

Cette partie est une évaluation des implémantations que nous avons mises en oeuvre pour réaliser la parallélisation de SystemC à l'aide de `pThreads`. Ainsi, on comparera les versions parallèles avec celle d'origine qui est purement coopérative.

1 Environnement des tests

Les tests ont été effectués sur 3 plateformes différentes :

- **Machine 1 : Machine Pentium(R) Dual-Core**
- **Machine 2 : Machine fema** : Machine Quad-Xeon 2.67GHz
- **Machine 3 : Machine ensibm** : 4 processeurs 8 coeurs Intel Xeon 75xx Nehalem EX (avec Hyper-threading et TurboBoost) : 32 coeurs physiques, 64 virtuels.

Ainsi on a pu avoir des machines multiprocesseurs avec un nombre de coeurs assez différent pour pouvoir évaluer l'efficacité des solutions de parallélisation de SystemC dont le but est justement d'exploiter les machines SMP actuelles.

2 Test d'affectation simple d'une variable

On dispose dans ce test d'un module avec un attribut `x` et un seul `SC_THREAD` qui ne fait qu'une affectation de la variable `x` à la valeur 2 durant 10.000 tours de boucle.

On instancie 150 objets du module pour avoir différents `SC_THREADS` s'exécutant en concurrence. Le but est tout simplement de voir l'efficacité des différentes solutions pour des `SC_THREADS` qui réalisent des tâches très minuscules.

On a alors obtenu les résultats suivants, correspondant au temps de simulation (exécution de `sc_start`) :

	Version séquentielle	**Sol. `1 SC_Thread↔1 pThread`**	**Sol. prod/cons**
Machine 1	`0.35 s`	`14.50 s`	`4.83 s`
Machine 2	0.22 s	6.36 s	1.58 s
Machine 3	0.18 s	21.72 s	7.25 s

FIGURE 6 – Résultats des tests d'affectation de variable

On remarque alors que la solution naïve est beaucoup plus coûteuse que la version séquentielle. Alors que la solution avec producteur consommateur reste à peu près plus proche de l'ordre de grandeur que la version séquentielle.

Ceci s'explique par le fait qu'on avait créé plusieurs pthreads dans la solution naïve. Or cette opération n'est rien d'autre que des appels système assez coûteux et qui nécessitent des changements de contexte à chaque fois. Ainsi, on ne gagne pas de temps dans ce test qui, lui, consite simplement à exécuter des affectations de variable. La solution avec producteur/consomateur, quant à elle, définit dès le début le nombre de `pthreads` créés et donc exploite le parallélisme de la machine de façon plus intelligente.

3 Fractale de Mandelbrot

On considère la suite (u_n) définie par la relation de récurrence :

$$\begin{cases} u_{n+1} = u_n^2 + z_0 & ,\forall n \in \mathbb{N} \\ u_0 = 0 \end{cases}$$

L'ensemble de Mandelbrot est l'ensemble des complexes z_0 tels que cette suite ne diverge pas.

On essaie alors de savoir la "vitesse de divergence" de la suite. On fixe un réel $u_{max} \succ 2$. On distingue alors deux cas :

- Il existe m_0 tel que $|u_{m_0}| \geq u_{\max}$.
- sinon, la suite ne diverge pas.

La couleur alors du pixel dépend de l'entier m_0, sinon elle est noire en cas de convergence.

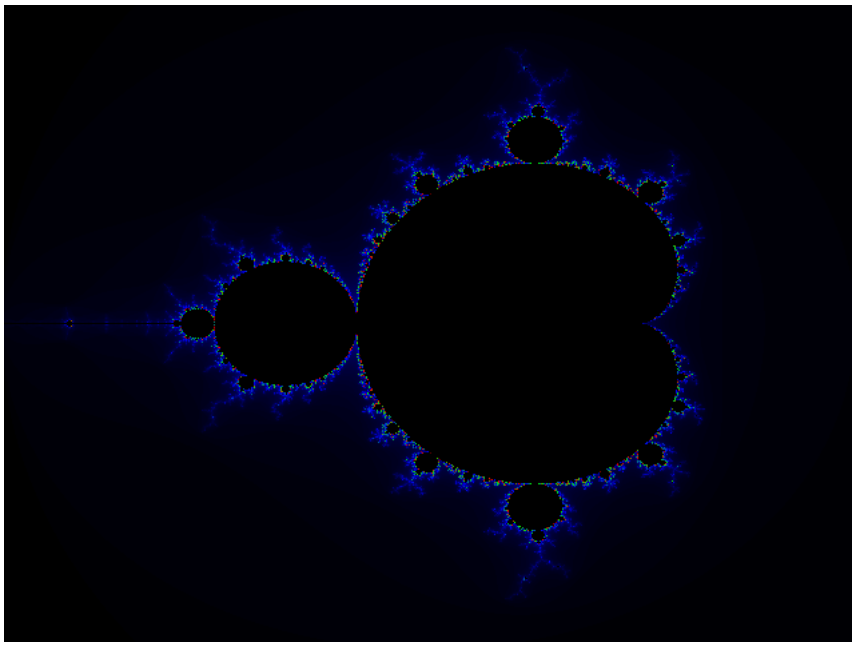
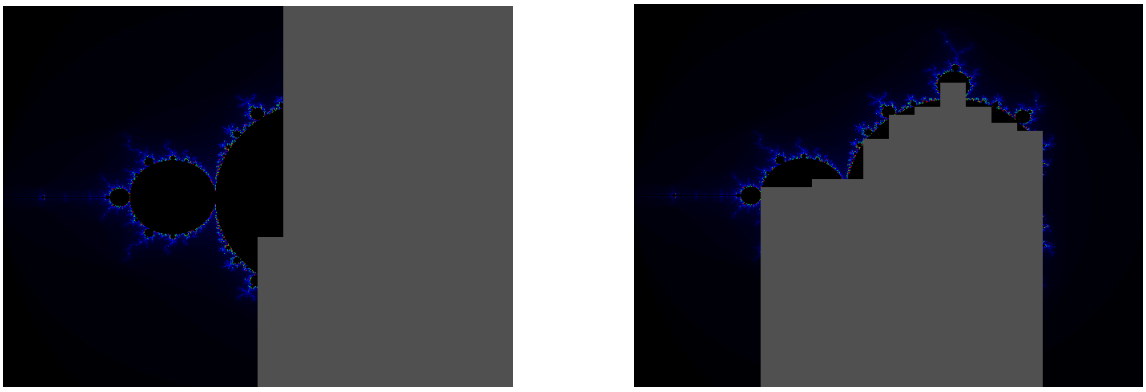


FIGURE 7 – Fractale de Mandelbrot

Ainsi, le dessin de cette fractale peut être divisé en des "slices" verticales.

On aura dans notre cas des `SC_Threads`, qui réalisent tous la même tâche consistant à dessiner le prochain slice.

La parallélisation du dessin consistera alors à avoir des threads qui chacun calcule indépendamment toute une ligne verticale. Le schéma suivant illustre le dessin dans les deux versions, séquentielle et parallèle.



(a) (b)

FIGURE 8 – Dessin de la fractale de Mandelbrot de façon séquetielle (a) et parallèle (b)

Contrairement, au test précédent, le dessin de chaque slice de la fractale de mandelbrot demande un calcul assez consistant en itérant à chaque fois pour trouver la couleur correspondante.

Les tests effectués sur les différentes machines sont récapitulés dans le tableau suivant :

	Version séquentielle	**Sol. `1 SC_Thread↔1 pThread`**	**Sol. prod/cons**
Machine 1	`13.71 s`	`7.78 s`	`7.08 s`
Machine 2	10.83 s	3.84 s	3.56 s
Machine 3	9.78 s	1.74 s	1.39 s

FIGURE 9 – Résultats des tests de la fractale de Mandelbrot

On remarque alors que la parallélisation dans les deux cas est beaucoup mieux que la version séquentielle vu que le temps de simulation a fortement diminué. Ainsi on peut voir que pour des processus SystemC qui exécutent des parties de code assez grandes bénéficient d'un gain de temps très considérable que ça soit avec la solution naïve ou producteur/consommateur. En effet, on a un ratio d'accélération qui est proche du nombre de processeurs que possède la machine.

Il est clair aussi que la solution avec producteur/consommateur reste même plus rapide que la solution naïve dans tous les cas.

Cinquième partie
Conclusion

Le travail de ce TER était alors de trouver un moyen afin de pouvoir effectuer des simulations SystemC sur des machines multiprocesseurs.

Beaucoup de travaux ont déjà été réalisés dans plusieurs laboratoires et entreprises pour essayer d'arriver à une parallélisation de SystemC. Notre approche à nous était simple et consistait à utiliser la bibliothèque Threads POSIX et donc d'exploiter la préemptivité de l'ordonnanceur du système d'exploitation.

Ainsi nous avons présenté deux solutions. Une solution naïve consistant à associer à chaque `SC_THREAD` un `pThread`. Mais lors de la phase des tests, il s'est avéré que si les processus SystemC n'effectuent pas des tâches lourdes, alors cette solultion peut devenir même plus coûteuse que la version séquentielle. Ceci est dû au coût des appels système lors de la création des différents threads POSIX.

Nous avons alors pensé à limiter le nombre de threads POSIX de le définir une fois pour toute au tout début. Choisir un nombre de threads POSIX égal au nombre de processeurs de la machine incrémenté de 1 pourra être une bonne valeur pour garantir une parallélisation efficace. Ainsi, on avait un modèle producteur / consommateur où les consommateurs sont des `pThreads` qui lisent sur un tampon dans lequel les différents `SC_THREAD` ont écrit la fonction réalisant tout ou une partie de leur code. Cette solution s'est avérée alors efficace dans tous les cas et quel que soit la taille du code que doit exécuter un processus SystemC. En fait, le ratio d'accélération est très proche du nombre de processeurs dont dispose la machine.

Comme on l'avait déjà mentionné, beaucoup d'autres travaux ont été réalisés sur ce sujet. On cherche toujours à avoir une solution élégante que des entreprises comme STMicroelectronics pourraient accepter et utiliser facilement. Un travail à Verimag dans le cadre d'un master Recherche consiste à modifier l'algorithme du scheduler SystemC afin de lui ajouter de la préemptivité.

ANNEXES

During.h

```
1  #ifndef __DURING_H__
2  #define __DURING_H__
3
4  #include <systemc.h>
5  #include <pthread.h>
6  #include <semaphore.h>
7
8  #define TAILLE_TAMPON 3
9
10 void during_threads(double time_ns, sc_time_unit unit, void *(*routine) ↩
       (void *), sc_core::sc_module *m);
11
12 void init_prod_cons();
13 void during_prod_cons(double time, sc_time_unit unit, void *(*routine) (↩
       void *), sc_core::sc_module *m);
14 void clean_threads();
15
16 #endif
```

During.cpp

```
1  #include "during.h"
2  #include <vector>
3
4
5  using namespace std;
6
7  //variable pour la solution threads
8  void *st;
9
10
11 //variable pour la solution prod/cons
12 typedef void *(*thread_run) (void *);
13
14
15 typedef struct {
16     thread_run routine;
17     sc_core::sc_module *m;
18 } mod_str;
19
20 mod_str *TAMPON;
21 sem_t mutex, plein, vide;
22 int indProd = 0, indCons = 0;
23
```

```
24 pthread_t *tabpthr;
25
26
27
28 /*Solution naÃ¯ve*/
29 void during_threads(double time_ns, sc_time_unit unit, void *(*routine) ↩
       (void *), sc_core::sc_module *m)
30 {
31     pthread_t thr;
32     void * sta;
33     pthread_create(&thr, NULL, routine, m);
34     wait(time_ns, unit);
35     pthread_join(thr, &sta);
36 }
37
38 //-------------------------------------------------//
39
40 /*solution avec prod/cons*/
41 void ecrire_tampon(mod_str fct)
42 {
43     sem_wait(&vide);     //vide--
44     sem_wait(&mutex);    //dÃ©but de section critique
45     TAMPON[indProd] = fct;
46     indProd = (indProd + 1) % TAILLE_TAMPON;
47     sem_post(&mutex);    //fin de section critique
48     sem_post(&plein);    //plein++
49 }
50
51 void* lire_tampon(void *arg)
52 {
53     while (true) {
54         mod_str ret;
55         sem_wait(&plein);    //plein--
56         sem_wait(&mutex);    //dÃ©but de section critique
57         ret = TAMPON[indCons];
58         indCons = (indCons + 1) % TAILLE_TAMPON;
59         sem_post(&mutex);    //fin de section critique
60         sem_post(&vide);     //vide++
61         ret.routine(ret.m);
62     }
63
64     return NULL;
65 }
66
67 void during_prod_cons(double time, sc_time_unit unit, void *(*routine) (↩
       void *), sc_core::sc_module *m)
68 {
69     mod_str ms = {routine, m};
70     ecrire_tampon(ms);
71     wait(time, unit);
72 }
73
```

```
74 void init_prod_cons()
75 {
76     TAMPON = new mod_str[TAILLE_TAMPON];
77     tabpthr = new pthread_t[TAILLE_TAMPON];
78 
79     sem_init(&mutex, 0, 1);
80     sem_init(&vide, 0, TAILLE_TAMPON);
81     sem_init(&plein, 0, 0);
82 
83     //instanciation des consommateurs une fois pour toute
84     for (int i = 0; i < TAILLE_TAMPON; i++)
85         pthread_create(&tabpthr[i], NULL, lire_tampon, NULL);
86 
87 }
88 
89 
90 void clean_threads() {
91     for (int i = 0; i < TAILLE_TAMPON; i++)
92         pthread_cancel(tabpthr[i]);
93 }
```

Références

[1] Yussef Bouzouzou. Accélération des simulations de systèmes sur puce au niveau transactionnel. Diplôme de recherche technologique, Université Joseph Fourier, 2007.

[2] Yves Denneulin et Jacques Mossière. *Cours SEPC*. Grenoble INP - ENSIMAG, 2010.

[3] Giovanni Funchal and Matthieu Moy. jTLM : an experimentation framework for the simulation of transaction-level models of systems-on-chip. In *Design, Automation and Test in Europe (DATE)*, 2011.

[4] OSC Initiative. *IEEE Standard SystemC Language Reference Manual*, 2006.

[5] Wen Yu Su. New parallel systemc simulation kernel. In *Scream Lab Presentation*, Taiwan, 2010.