IRL : Simulation distribuée pour les systèmes embarqués

2014

Amaury Graillat, 2ème année Ensimag, Grenoble INP
Matthieu Moy, Verimag

Introduction

SystemC est une bibliothèque C++ de description matérielle au niveau TLM (Transaction-level modeling), c'est-à-dire que chaque partie d'un circuit est représentée par un objet (mémoire, processeur...). Cette modélisation ne rentre pas dans les détails des signaux mais uniquement de l'information qui transite. Cela offre deux avantages : la simulation est plus rapide qu'avec des langages de plus bas niveau tels que VHDL ou Verilog et surtout, elle permet de tester le logiciel avant même de connaitre l'implémentation matérielle.

```
SC_MODULE(PorteOu) {
        sc_out<bool> s;
        sc_in<bool> e1, e2;

        void compute() { s.write((e1.read() || e2.read())); }

        SC_CTOR(PorteOu) {
                SC_METHOD(compute);
                sensitive << e1 << e2;
        }
};
```

FIGURE 1 – SystemC : porte OU

La figure 1 donne l'exemple d'une porte OU en SystemC. Le composant est représenté par un module. La méthode `compute` est exécutée par SystemC a chaque modification des entrées e1 et e2. Au changement d'une entrée, SystemC va réévaluer tous les signaux séquentiellement jusqu'à la convergence.

Problématique Fondamentalement, les circuits décrits par SystemC sont parallèles, car ils sont composés de différents composants qui, physiquement, fonctionnent en parallèle. En terme de performances, le principal problème est que SystemC exécute la simulation de manière séquentielle. SystemC ne bénéficie ainsi pas des performances offertes par les processeurs multi-cœurs pour accélérer son exécution.

Si on considère chaque module comme un processus, rendre SystemC physiquement parallèle pourrait, à première vue, paraitre évident en parallélisant chaque processus. Cependant cela obligerait à disposer d'un algorithme permettant de résoudre les dépendances de données entre les différents processus, ce qui en pratique est très complexe [1]. Nous choisissons une autre approche :

Sc-during Nous nous basons sur la bibliothèque sc-during [3] créée par Matthieu Moy. Elle permet à l'utilisateur de SystemC de préciser les parties du code qu'il souhaite rendre parallèles [1].

SystemC a un mode de fonctionnement dit *loosely-timed*, c'est-à-dire que les opérations exécutées sont considérées comme instantanées. Il n'y a aucun rapport entre le temps d'exécution de la simulation (*wall clock time*), et l'horloge du circuit simulé. Dans ce modèle, le temps est une variable qui doit être incrémentée explicitement. C'est le but de la fonction `wait` de SystemC.

Ainsi, ce mode de fonctionnement offre des libertés sur l'ordre et la date d'exécution de certaines opérations.

Sc-during fournit la fonction `during` qui sert à demander l'exécution d'une fonction pendant une certaine durée. Concrètement, l'ordonnanceur choisit d'exécuter cette fonction à n'importe quels instants de cette durée en lui créant un thread.

Dans la fig. 2, le calcul du module A demande l'exécution de f avec `during(f, durée)`. Le thread 1 est créé, ainsi B peut s'exécuter en parallèle. Donc, A et B, s'exécutent sur le thread principale de SystemC et f sur Thread 1.

Contribution Nous créons une version distribuée de `during` en utilisant la bibliothèque MPI.

Le principe de base de l'implémentation de `during` en systèmes distribués, est de lancer l'exécution d'une fonction sur une autre machine. Cependant, cela pose des problèmes que ne posait pas la version multi-threadée :

1. Nous ne pouvons pas partager de structures de données entres les machines.
2. Les différentes entités sont totalement asynchrones et ne peuvent être synchronisées que par message.

1. Il existe une autre approche [2] de distribution semi-automatisée. Le programmeur spécifie la fréquence d'évaluation des sorties de chaque modules. La simulation de chaque module est donc exécutée de manière asynchrone jusqu'au prochain instant probable de modification de ses entrées.

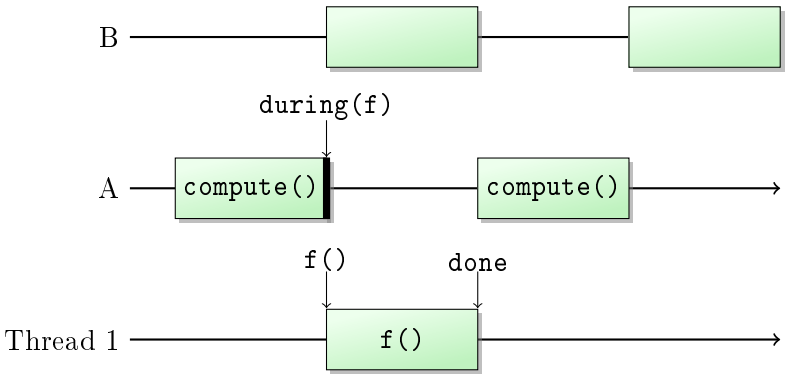


FIGURE 2 – Sc-during : méthode during

Notre solution à ces problèmes consiste à disposer d'un protocole de communication pour dialoguer entre une machine principale, qui exécute SystemC, et des machines secondaires qui exécutent les fonctions en parallèle.

Nous ne proposons pas de moyen de partager les paramètres et les données entre les différentes machines, mais cela serait réalisable en étendant le langage que nous proposons.

Nous parlerons des prérequis et des bibliothèques utilisées, ensuite des modifications apportées à la bibliothèques sc-during et enfin des performances de cette approche.

1 Prérequis : SystemC, MPI et sc-during

1.1 SystemC

Un programme SystemC est composé de `SC_MODULE` qui sont des classes C++.

SystemC propose deux types de processus qui ont des manières différentes d'exécuter une fonction.

`SC_METHOD(f)` permet de définir des signaux de réveil. La fonction est relancée à chaque changement sur les signaux.

`SC_THREAD(f)` exécute une seule fois la fonction, le contexte est sauvegardé (donc l'exécution reprend où elle s'était arrêtée). Il faut donc utiliser `wait` pour l'arrêter.

L'ordonnanceur de SystemC exécute tous les processus dans un ordre arbitraire. Cet ordonnanceur étant collaboratif, les tâches rendent la main explicitement : soit par un `wait`, soit à leur terminaison.

1.2 MPI

Message Passing Interface un standard de communication pour la programmation distribuée. Il fournit une interface d'envoi et de réception de messages entre machines et offre différents modes de communications. Nous utilisons le mode point-à-point.

L'initialisation est fait par la fonction `MPI_Init` qui utilise les arguments de l'exécutable (`MPI_Init(&argc, &argv)`) pour communiquer avec `mpiexec`, le lanceur MPI.

Ainsi, MPI se charge d'exécuter un même programme sur différentes machines et leur assigne un numéro unique (`rank`).

La communication entre machines se fait à l'aide des fonctions `MPI_Recv` et `MPI_Send` qui, respectivement, envoi un message et attend la réception d'un message. `MPI_Send` prend en paramètre le `rank` du destinataire.

MPI propose des messages dans différents formats. Nous nous contentons d'envoyer et de recevoir des tableaux d'entiers (`MPI_INTEGER`).

1.3 Sc-during

La bibliothèque sc-during fournit des fonctions de parallélisation pour SystemC. Elle ne nécessite pas de modification de SystemC et ainsi peut fonctionner sur toutes les implémentations propriétaires.

Pour utiliser sc-during, un SC_MODULE doit hériter de la classe sc-during. Cette dernière fournit les fonctions décrites précédemment (`during`, `catch_up` et `extra_time`).

during Lors de l'appel de `during`, un thread est créé. Le thread lance la fonction à paralléliser. A la fin de la fonction, le thread indique par l'intermédiaire d'un sémaphore et d'une variable sa terminaison. La fonction `during` attend la terminaison de la fonction exécutée, et rend la main à SystemC.

extra_time Permet d'allonger la durée initialement demandée à `during`. On peut la considérer comme l'équivalent d'un `wait`. L'appel de `extra_time` est effectué par la tâche exécutée par le thread. La fig. 3 montre une tâche demandant un temps t2 supplémentaire.

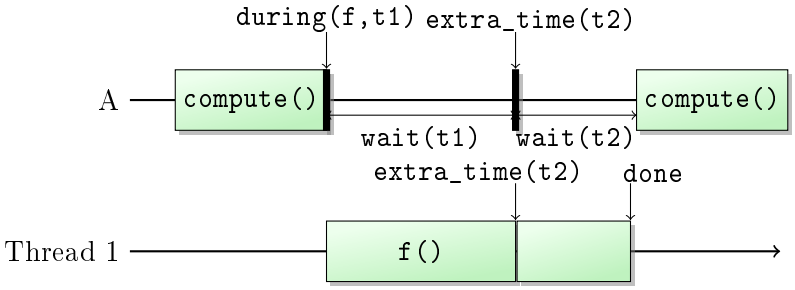


FIGURE 3 – Sc-during : méthode `extra_time`

2 Adaptation de sc-during pour les systèmes distribués

Le fonctionnement de sc-during (décrit en 1.3) n'est pas adapté à une architecture distribuée car il utilise des sémaphores et des variables qui ne peuvent pas être partagés avec des machines distantes. Nous avons donc développé une architecture à base de messages.

2.1 Architecture globale

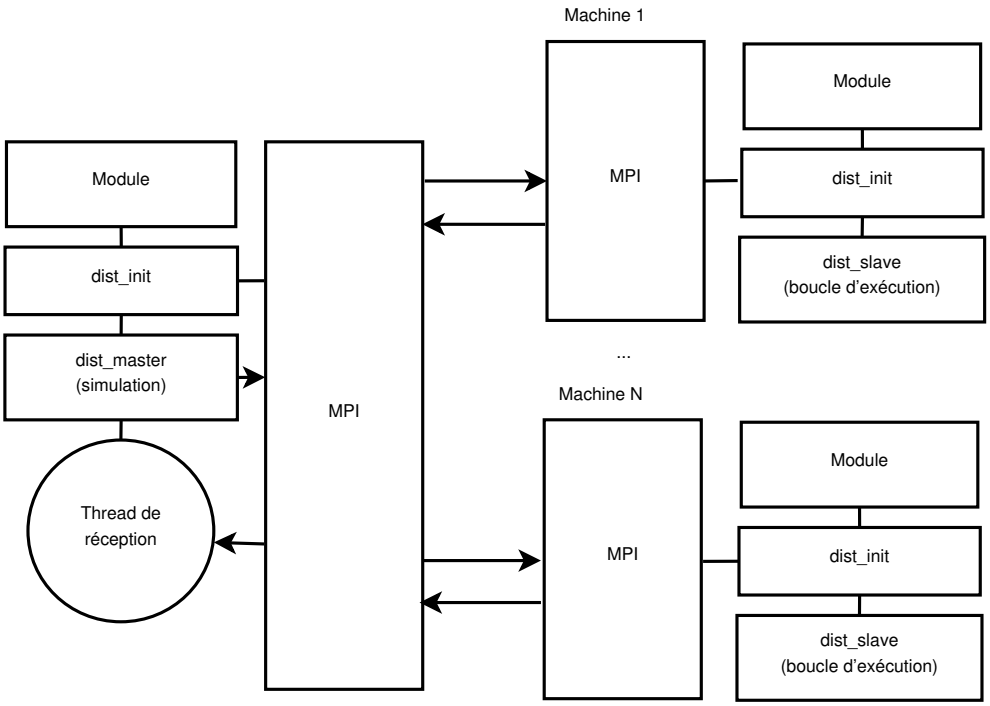


FIGURE 4 – Sc-during : méthode during

MPI associe un numéro unique aux différentes instances du programme. Nous définissons deux comportements en fonction du numéro. Le zéro est appelé Maitre, et les autres sont appelés Machines (cf. fig. 4).

Machines Une machine a pour but d'attendre et d'exécuter les commandes du Maitre. Elle doit exécuter les fonctions demandées, et envoyer un signal à la terminaison. Dans le cas d'un `extra_time`, elle se contente d'envoyer un message au maitre.

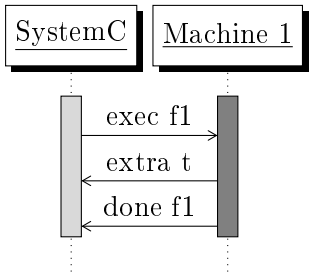
Maitre Le maitre à pour but d'exécuter le programme et d'envoyer les commandes aux machines pour permettre la parallélisation. Le maitre dispose d'un thread de communication lui permettant de recevoir les résultats envoyés par les Machines. Il en a besoin car on peut recevoir une terminaison de tâche pendant que SystemC est en train de simuler.

2.2 Protocole de synchronisation et de distribution des tâches

exec	idTask	fNum
done	idTask	-
extra_time	idTask	t

idTask correspond au numéro de tâche et de machine (on autorise une seule tâche à la fois par machine). Le dernier paramètre varie en fonction de la commande. Pour `exec`, il s'agit du numéro de la fonction à exécuter et pour `extra_time`, la durée de la prolongation.

La mise en œuvre de `during` se fait à l'aide d'une commande `exec` envoyée par le Maitre à une Machine. À la terminaison, cette dernière répond par un `done` avec la même valeur de idTask.



Ce diagramme montre une machine exécutant une fonction f1. Cette fonction demande du temps supplémentaire avec `extra_time`. À sa terminaison, elle envoie un message `done`. La machine ne dispose pas des données pour traiter `extra_time`, elle envoie donc un message au maitre qui exécute la fonction `extra_time` fournit par sc-during.

2.3 Implémentation

Pour implémenter `during`, il a fallu trouver une méthode pour identifier les fonctions dans les communications. Nous n'avons pas de garantie sur l'égalité des adresses mémoire des fonctions entre le maitre et les machines. Ainsi nous avons créé la fonction `registerFunction` pour assigner un entier à chaque fonction, et stocker leur références dans un tableau.

L'objet `dist_init` fournit les fonctions communes au maitre et aux machines comme l'initialisation de MPI et l'enregistrement des fonctions. En fonction du rank de la machine (0 pour le maitre), la fonction initCommunication instancie soit un objet `dist_master` soit `dist_slave`.

`dist_master` contient la fonction du thread d'écoute des messages et des fonctions d'assignation des machines libres (esquisse d'ordonnanceur).

`dist_slave` contient une fonction d'écoute des messages et d'exécution des fonctions.

Les fonctions `during` et `extra_time` distribuées sont dans une classe `sc_during_dist`.

```
SC_MODULE(A), sc_during
{
    SC_CTOR(A) { SC_THREAD(compute); }

    void compute() {
        during(10, SC_NS, 0); // f0
    }

    static void f() { /* calcul */ }
};

int sc_main(int argc, char *argv[])
{
    dist_init& di = dist_init::getInstance();
    di.registerFunction(&A::f); // f0
    di.initCommunication(argc, argv);

    A a("a");
    sc_start();

    return 0;
}
```

FIGURE 5 – Exemple d'utilisation de during distribué

Le code fig. 5 donne un exemple d'utilisation de `during` distribué. Une fonction f est exécutée sur une machine distante. On note que registerFunction lui associe le numéro 0 qui doit être réutilisé pour l'appel de during.

3 Résultats

3.1 Courbes de performance

Pour comparer les performances avec l'implémentation basée sur les threads de sc-during, nous avons créé un programme contenant 30 modules effectuant des calculs flottants. Le temps de calcul séquentiel est de 41 secondes. La machine de test dispose de 16 processeurs de 2Ghz hyperthreadés (Intel(R) Xeon(R) CPU E5-2650).

Sur cet exemple nous avons observé (voir fig. 7) des performance proches mais souvent inférieurs à la version threadé. Cela vient de la lourdeur de MPI comparée à la création d'un thread dans un système d'exploitation.

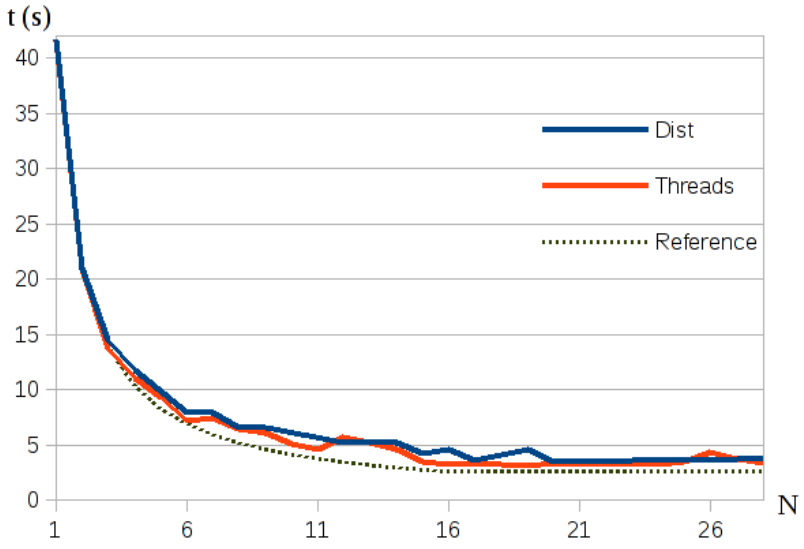


FIGURE 6 – Comparaison des performance de during multi-threadé et distribué

La figure 7 montre une accélération proche de la référence pour un petit nombre de cœur et inférieure ensuite. Une analyse détaillée serait nécessaire pour définir l'origine de la perte de performance.

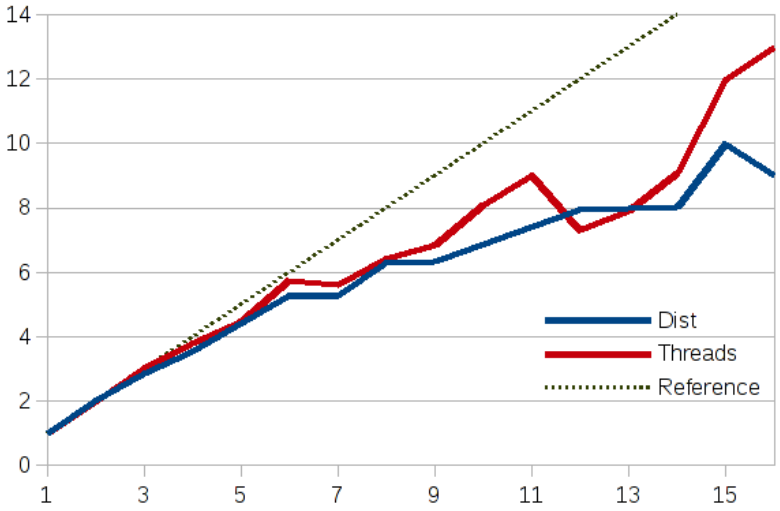


FIGURE 7 – Accélération en fonction du nombre de machine

Ces mesures de performances ne sont pas représentative d'une utilisation réelle de MPI car elle a été réalisée sur une seule machine, annulant ainsi les temps de latence réseau.

3.2 Suite du travail et limitations

Nous n'avons pas eu le temps de traiter la terminaison des programmes sur chaque machine distribuée. Cela nécessiterait de s'assurer que toutes les tâches ont été exécutées avant de terminer chaque machine et le maitre.

Un suite possible du travail serait de permettre l'appel des fonctions distantes avec des paramètres, et la récupération du résultat du calcul.

Il existes aussi d'autres fonctions de Sc-during (`sc_call`, `catch_up`, etc), que nous n'avons pas adapté.

Conclusion

Notre travail a été de trouver une approche pour rendre une simulation SystemC distribuable. Nous avons utilisé MPI pour les communications. Nous avons redéfini les fonctions `during` et `extra_time` de la bibliothèque sc-during.

La simulation est exécutée par un maitre. Il est connecté à un certain nombre de machines qui exécutent des fonctions sur demande. `during` a été implémenté par un envoie de message à une machine choisi arbitrairement parmi les machines libres. Lorsqu'elle a terminé d'exécuter, la machine l'indique au maitre par l'intermédiaire d'un message.

L'implémentation de `extra_time` est un simple envoi de message d'une machine vers le maitre qui appel la méthode `extra_time` fournit par sc-during.

Sur des exemples simples, nous avons montré qu'il était possible de distribuer la simulation. Cependant, il s'agit seulement des prémices du travail car il n'est pas encore possible de récupérer le résultat des calculs ni d'utiliser toutes les méthodes fournit par sc-during. Cela pourrait constituer une suite de cette recherche.

Références

[1] Yussef BOUZOUZOU. Acceleration des simulations de modeles de systèmes sur puce au niveau transactionnel (7.1.4. Identification des problèmes d'efficacité de la parallélisation). 2007.

[2] Digital Force Mario Trams. Conservative Distributed Discrete Event Simulation with SystemC using Explicit Lookahead. 2004.

[3] Matthieu Moy. Parallel Programming with SystemC for Loosely Timed Models : A Non-Intrusive Approach. 2012.