Test d'Intel Parallel Studio

par Matthieu Brucher (site) (Blog)

Date de publication : 20 mai 2009

Dernière mise à jour : 2 juillet 2009

J'ai testé le dernier-né des outils Intel, **Parallel Studio**.

Introduction

Intel Parallel Studio est un ensemble d'outils dédiés à l'optimisation des programmes multithreadés. Il s'agit de plusieurs plugins de l'environnement Visual Studio. Il est donc nécessaire de posséder ce dernier (attention, la version Express ne supporte pas les plugins).

On trouvera donc les plugins suivants, pouvant aussi être achetés séparément :

- Parallel Inspector, analysant la gestion de la mémoire
- Parallel Amplifier, analysant le comportement des threads
- Parallel Composer, contenant l'extension parallèle pour le débuggeur

I - Les outils connus

I-A - Le compilateur C++ 11.1

La version 11.1 du compilateur d'Intel propose de nouvelles fonctionnalités :

- OpenMP 3.0
- Intégration de TBB et IPP

Je ne vais pas faire une analyse complète de ce compilateur, cela peut se résumer à d'excellentes performances, des extensions utiles (dont certaines parallèle) et un support de la norme C++0x telle qu'elle est définie actuellement (par exemple les fonctions lambda).

I-B - Threading Building Blocks

Il s'agit d'un ensemble d'outils permettant de découper un travail en bloc, chaque bloc étant envoyé sur un thread. La répartition est effectuée par un ordonnanceur.

I-C - Integrated Performance Primitives

Les outils de traitement du signal se trouvent dans IPP. Il s'agit de fonctions non standard (contrairement à celles proposées dans la MKL, la bibliothèque scientifique optimisée d'Intel, malheureusement absente) pour effectuer des traitements 1D, 2D, sur des vidéos, ...

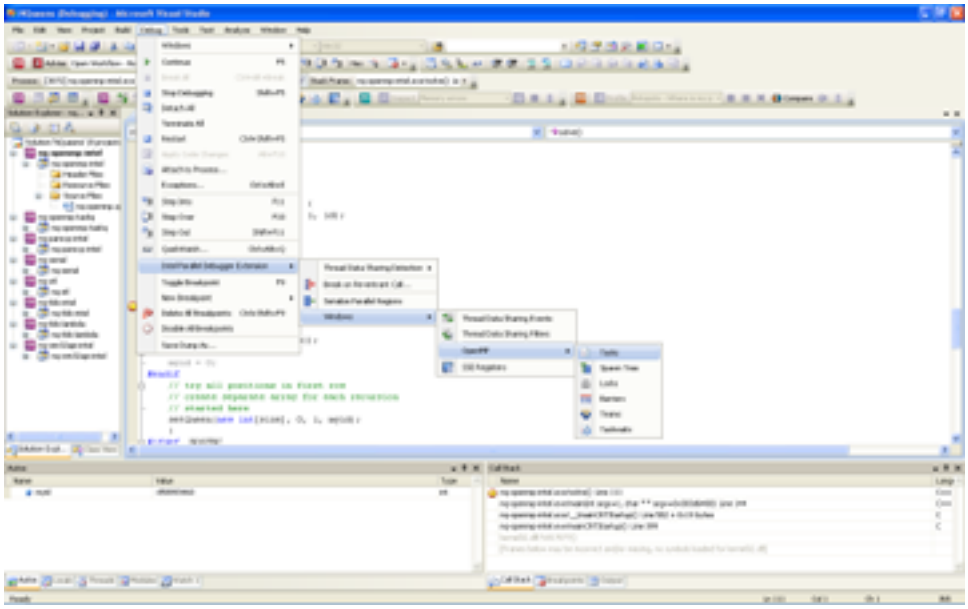
II - Les nouveaux outils

Chacun des 3 plugins du Parallel Studio peut s'utiliser séparément. Intel conseille de commencer par Composer, puis Inspector, pour finir par Amplifier. Un autre outil en phase de développement, Advisor, a pour objectif de piloter la réflexion.

II-A - Parallel Composer

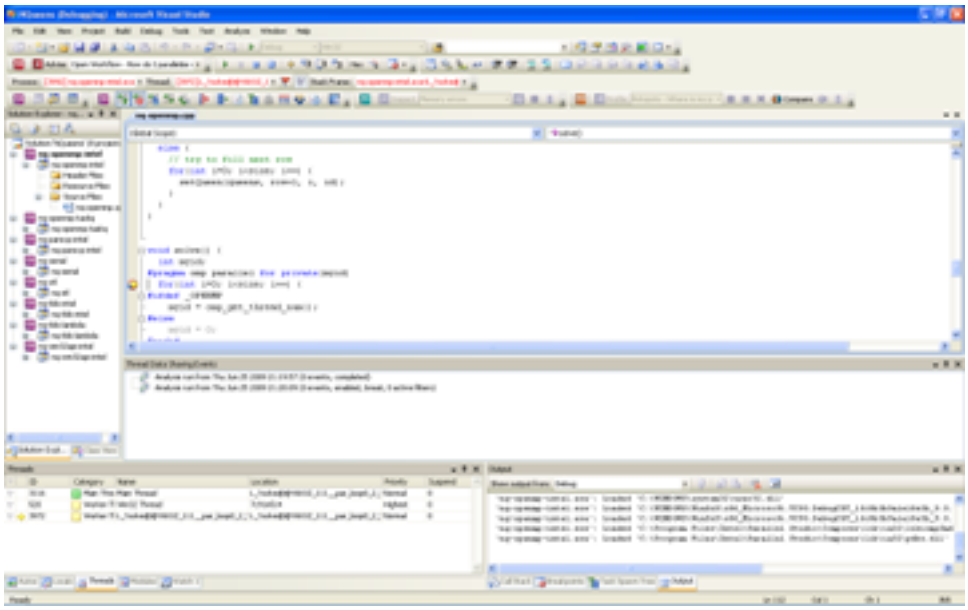
L'ajout principal de ce plugin est l'extension parallèle pour le débuggeur. C'est aussi avec ce plugin qu'est livré le compilateur Intel 11.1. Il y a donc des liens forts entre les deux. Pour activer l'extension, il est nécessaire d'indiquer que l'on désire détecter les événements parallèles (ce qui permet de ne pas toujours avoir ce surcoût lorsqu'on débuggue une application classique).

L'intérêt de cette extension n'est pas que dans la détection de données partagées entre threads, il s'agit aussi de vérifier des comportements dynamiques comme le code réentrant (est-ce qu'une fonction peut être appelée simultanément par plusieurs threads ?) ou l'arbre des tâches et des threads avec OpenMP.



Possibilités offertes par l'extension en mode debug

Les sous-fenêtres OpenMP ne sont pas que pour OpenMP. A partir du moment où l'option **/qopenmp** est utilisée (elle est nécessaire pour OpenMP, mais aussi pour les extensions parallèles du compilateur comme **__par**), elles peuvent afficher des informations utiles sur l'état des threads existants. De plus, il est alors possible de basculer en exécution monothread, ce qui permet aussi de vérifier le séquencement des exécutions.



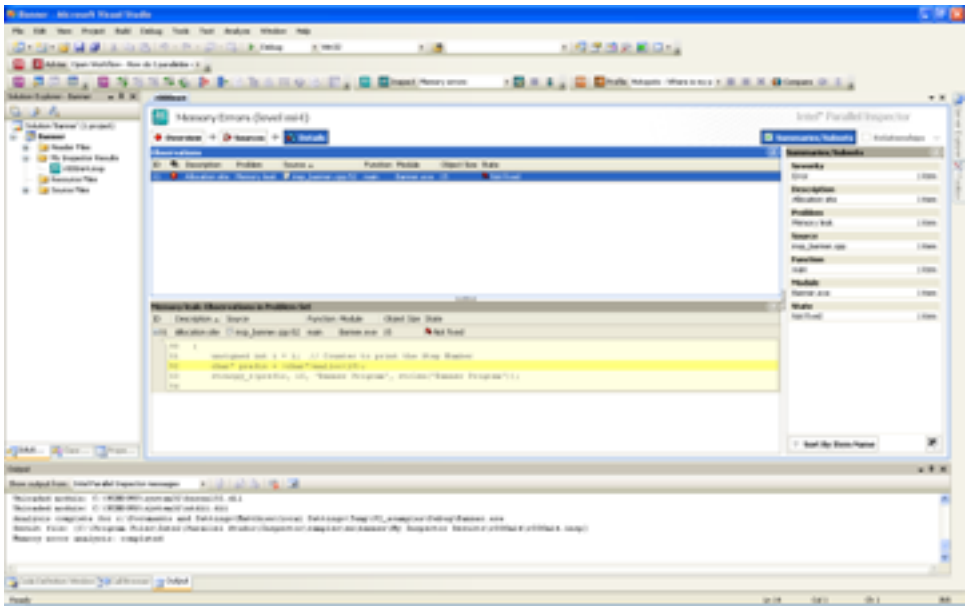
Différentes vues en cours de débuggage, comme l'arbre des tâches ou les événements

Il semblerait qu'il soit possible de débugger plusieurs processus simultanément, à la **TotalView**, mais aucun exemple ne le montre, aucune aide n'en parle.

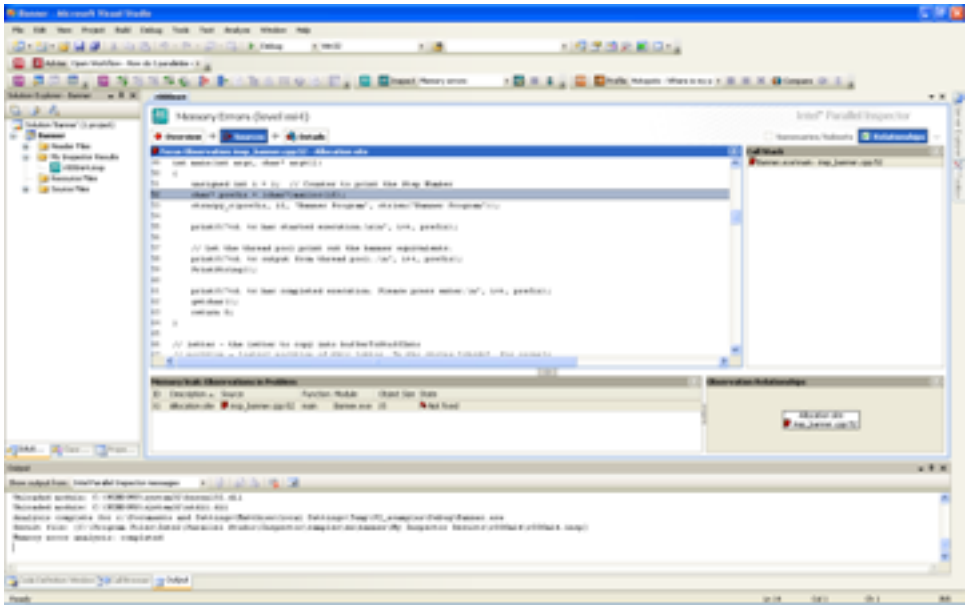
II-B - Parallel Inspector

L'inspecteur est chargé de détecter les problèmes de mémoire et de threads lorsqu'un programme fonctionne. Selon le niveau d'inspection, le temps d'exécution du programme peut être très long (mais on n'a rien sans rien). A chaque fois qu'un problème est détecté, sa gravité sera indiquée, l'emplacement de l'erreur et le code associé (qui pourra ensuite être modifié dans l'éditeur).

La première analyse possible est celle de la mémoire. Il est donc possible de vérifier, entre autres, les fuites mémoire. Le rapport reporte l'endroit d'allocation. Tout ceci est visible dans la capture d'écran suivante.

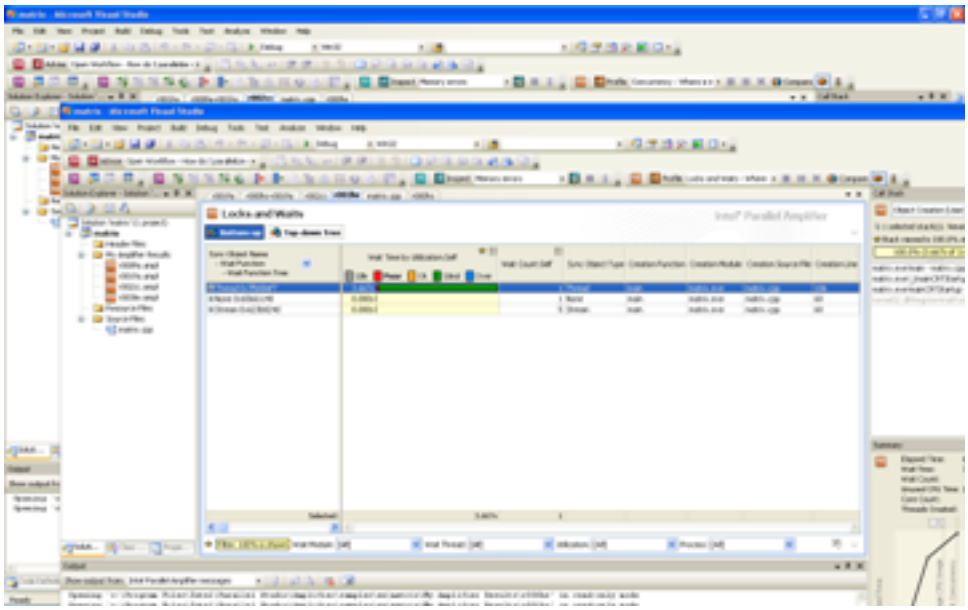


Rapport d'inspection mémoire

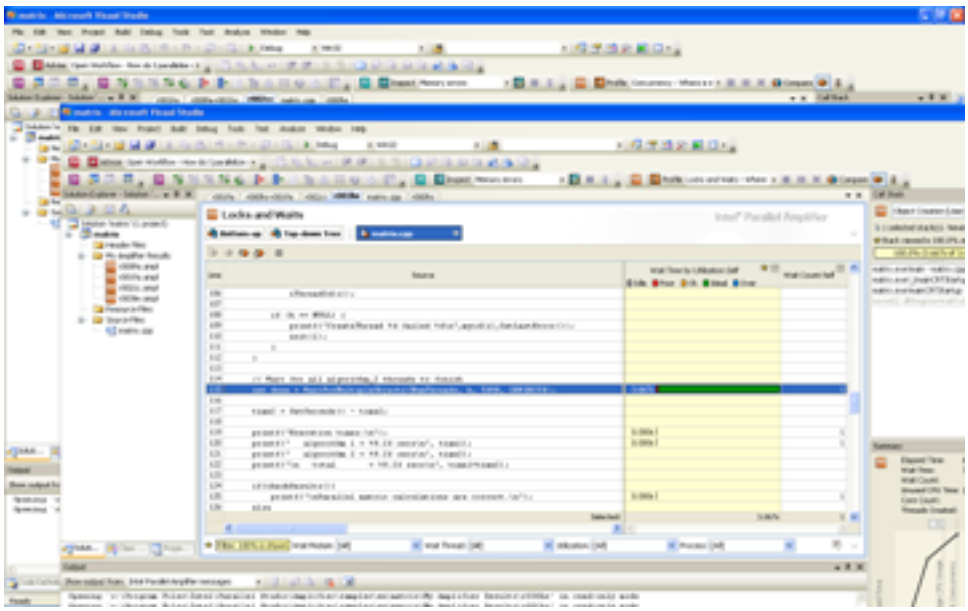


Le code source ayant généré une fuite mémoire

Parallel Inspector n'est pas Parallel pour rien. L'inspection des threads permet de vérifier les accès mémoire concurrents. Il est possible de supprimer certaines analyses afin d'accélérer l'inspection (mais seulement après qu'une première inspection ait été effectuée). Par exemple, si on connaît le comportement d'une variable, il n'est pas nécessaire de la surveiller.



Rapport d'inspection sur le parallélisme



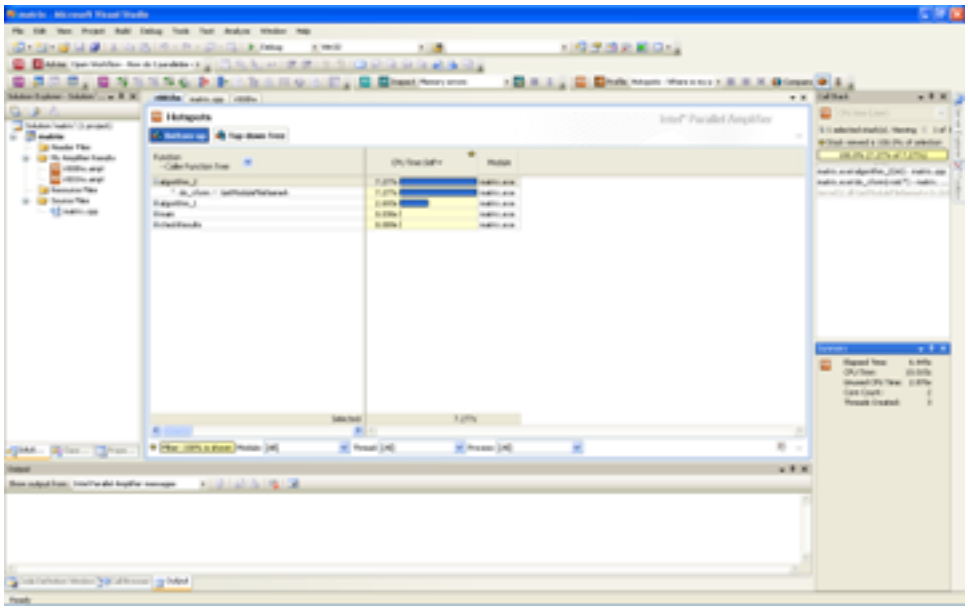
Affichage d'un problème d'accès en écriture à une même donnée

L'analyse mémoire est un problème récurrent. Il est parfois nécessaire d'ajouter une bibliothèque lors de la compilation ou de modifier son code. Ici, tout est fait en direct, ce qui est bien pratique. De même, l'inspection des threads est un outil qui peut être très utile lors d'un débuggage complexe où le multithreading intervient.

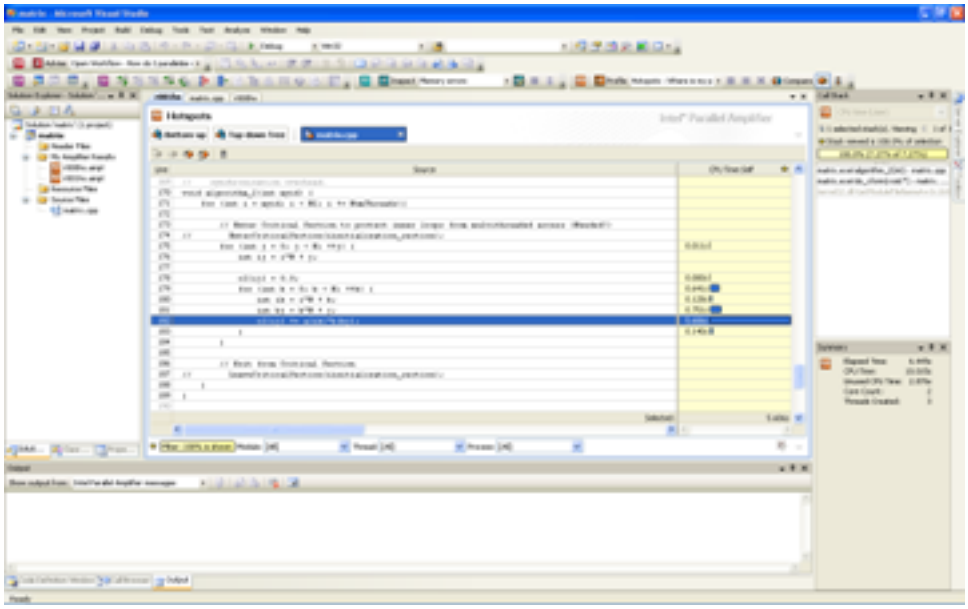
II-C - Parallel Amplifier

Amplifier effectue un profil du code à la volée. Le profil d'un code peut déjà être effectué par **certaines versions de Visual Studio**, mais avec plusieurs coûts différents. Ici, on se limitera à la durée de fonctionnement, à la qualité du parallélisme et aux délais d'attentes.

Le premier type de profil est le **hotspot**. Il s'agit de mesurer où l'application passe le plus clair de son temps. Dans le cas présenté, c'est la fonction **algorithm3** qui est la plus coûteuse. En double-cliquant dessus, on ouvre une fenêtre donnant ligne par ligne le pourcentage de temps passé.

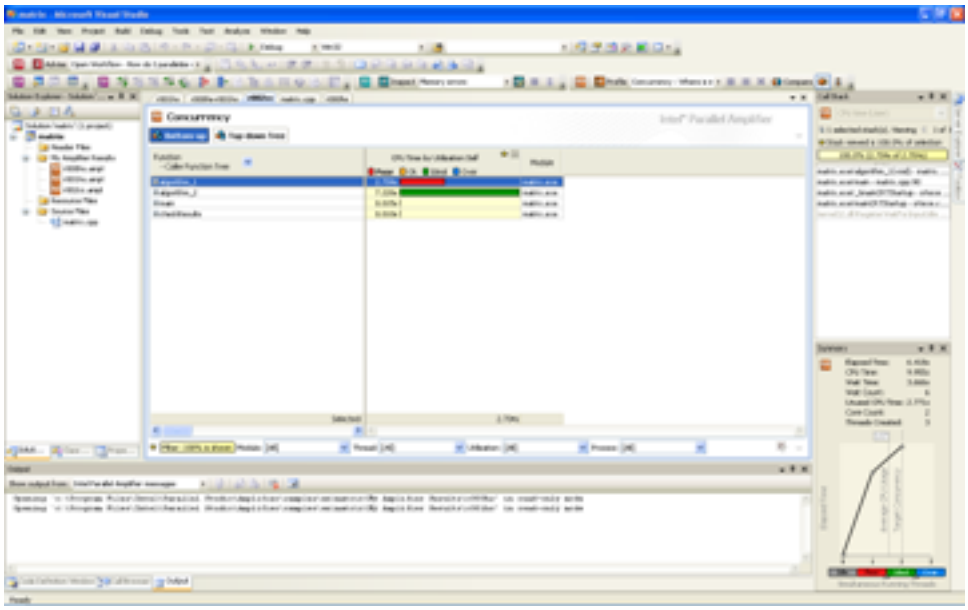


Détection des fonctions les plus coûteuses

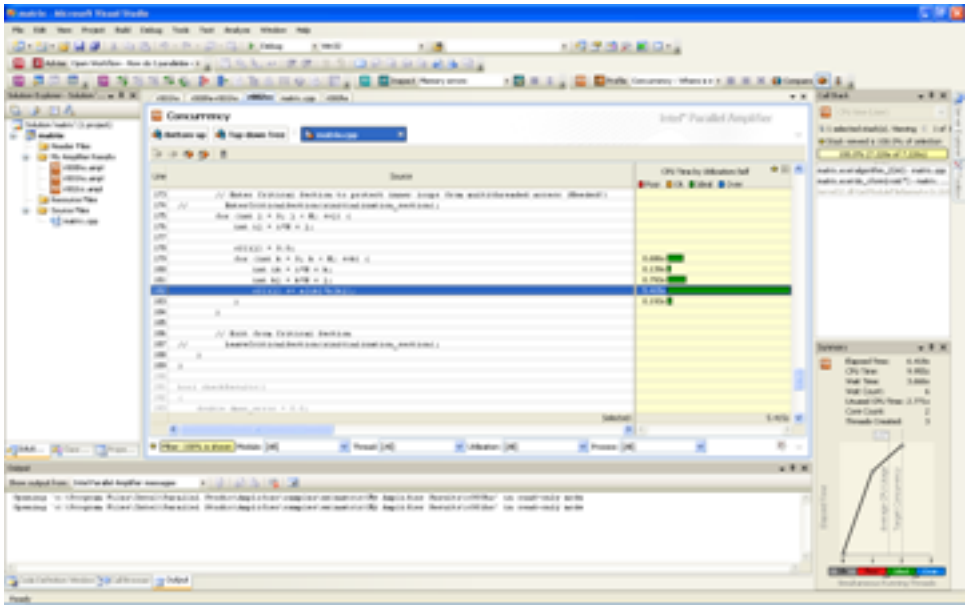


Détail des coûts par instruction

L'analyse du parallélisme, **concurrency**, indique, à partir du nombre de processeurs de la machine, la scalabilité de l'algorithme. Dans le cas présenté, on voit en bas à droite le gain total, pour deux processeurs qui est à 1.78, soit donc 88.8% d'utilisation. On peut demander le détail des sources qui indique que le manque de parallélisme vient surtout des routines d'affichage. En revanche, la fonction la plus coûteuse, *algorithm3*, en temps a un bon parallélisme.

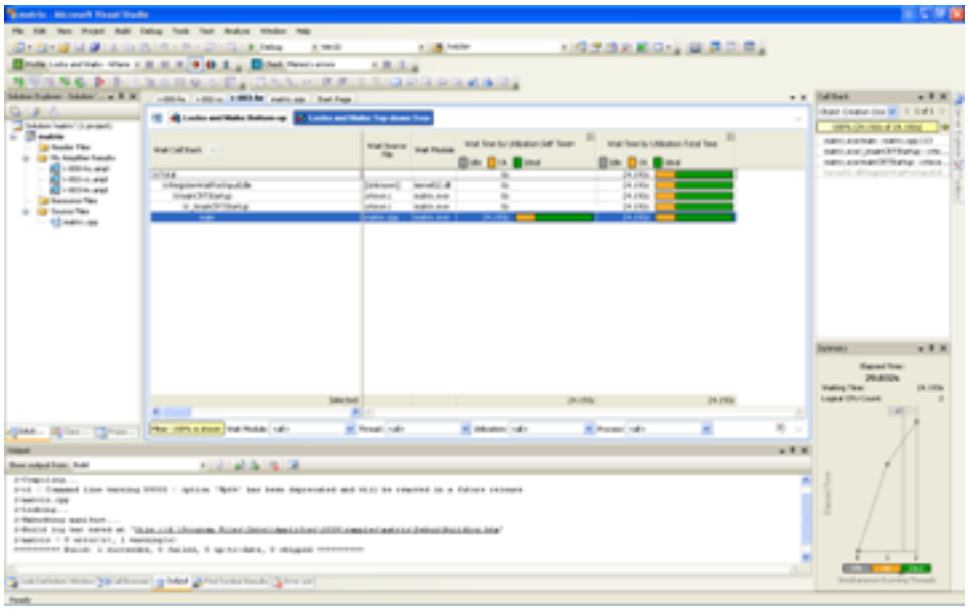


Détection du parallélisme sous-optimal

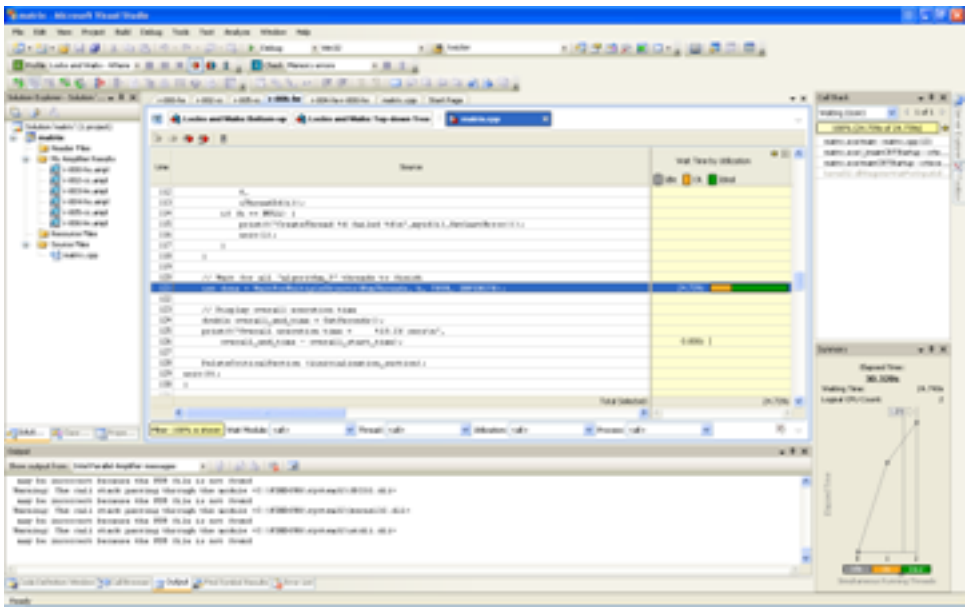


Détection du parallélisme sous-optimal

Enfin, un des gros problèmes du parallélisme concerne les attentes et les verrous. Ici encore, Amplifier propose une solution. Dans l'exemple ici, l'attente n'est présente que dans la fonction principale, lorsqu'elle attend que les sous-threads se terminent.

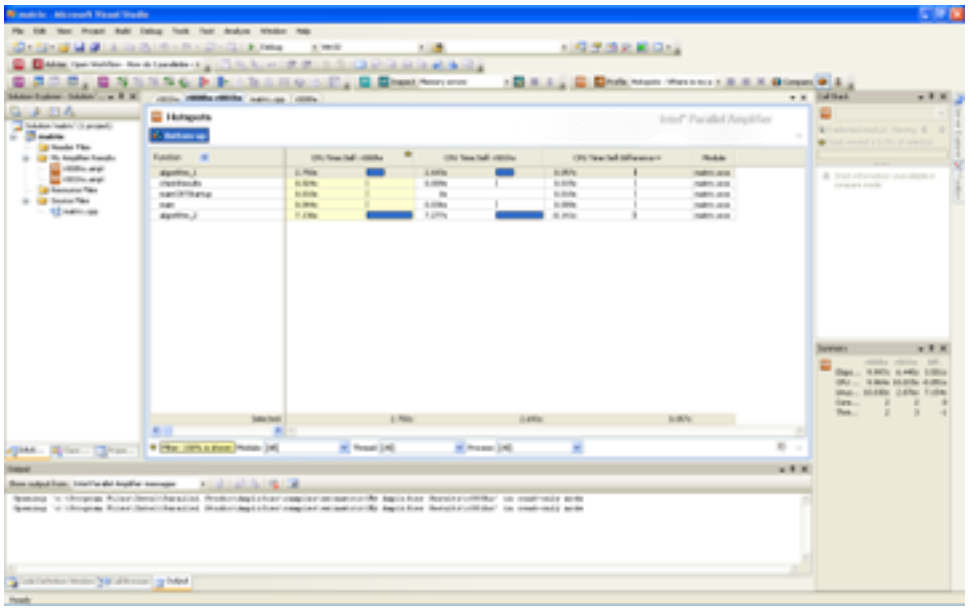


Détection des attentes et des verrous coûteux



Détail des verrous par instruction

Il est aussi possible de comparer les résultats entre plusieurs exécutions et changements. Cela permet de visualiser l'évolution des améliorations.



Comparaison entre deux exécutions hotspot

Les données retournées sont très pratiques et utiles (comme la vue sur le code source), avec une documentation en ligne claire et efficace.

Conclusion

Amplifier et Inspector sont intuitifs et simples d'utilisation, Composer l'est un peu moins. L'aide en ligne est facile d'accès (contrairement la version béta où tout était à faire), et Intel propose plusieurs vidéos sur son site Internet présentant les possibilités de Parallel Studio.

Parallel Studio est donc un produit qui me semble très utile, qui est peu envahissant dans son programme (je pense à Inspector et Amplifier pour la détection de certains problèmes sans bibliothèques additionnelles), et efficace. Les problèmes qu'il tente de résoudre ne sont pas faciles, et il s'en sort, à mon avis, avec brio.