Résumés de présentations

Parallélisation automatique de boucles sur processeurs vectoriels

Avec la fin imminente de la loi de Moore, il apparaît évident que les architectures courantes de processeurs sont appelées à changer. En fait, certains changements se sont déjà produit sans que la communauté en général ne s'en soit aperçu. Un exemple de ces changements est l'apparation de processeurs vectoriels promettant l'exécution simultanée de plusieurs intructions pour des gains de performances majeurs. Cette présentation abordera et fournira un début de réponse à la question: Comment puis-je accéder à ces gains de performance en faisant le moins d'effort possible? Pour ce faire, on s'attardera à l'architecture d'un compilateur moderne capable d'exploiter ce qui est en fait du parallèlisme au niveau des instructions.

Références Autovectorization

[1] Samuel P. Midkiff, *Automatic Parallelization - An Overview of Fundamental Compiler Techniques*, Morgan & Claypool Publishers, 2012

[2] Randy Allen & Ken Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*, Morgan Kaufmann, 2001

[3] Nadav Rotem & Arnold Schwaighofer, *Vectorization in LLVM*, LLVM Developpers Meeting, 2013

[4] Jim Radigan, *Inside Auto-Vectorization, 1 of n*, Channel 9, 2012

[5] Ayal Zaks & Dorit Nuzman, *Autovectorization in GCC-two years later*, Channel 9, 2012

[6] Ralf Karrenberg & Sebastian Hack, *Whole Function Vectorization*, Proceedings of the Ninth International Symposium on Code Generation and Optimization, 2011

[7] Saeed Maleki et al., *An Evaluation of Vectorizing Compilers*, Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques, 2011

Espace d'addressage global partitionné

Selon les estimations présentes de compagnies comme Cray et Intel, d'ici 2020, les superordinateurs devraient offrir des puissances de calcul de l'ordre de l'exaflop (10^{18} opérations en vigurle flottante à la seconde!) en utilisant environ 1 milliard (!) de *threads* pour y arriver. Malheureusement, les approches traditionnelles comme MPI ne satisfont plus la demande à ce niveau. Face à un tel défi, il nous faut de nouveaux outils. Dans cette présentation, je vous invite à découvrir un nouveau modèle de programmation conçu spécifiquement pour l'*exascale computing*: l'espace d'addressage global partitionné.

Références PGAS

[1] Hartmut Kaiser & Vinay Amatya, *HPX: A C++ Standards Compliant Runtime System For Asynchronous Parallel And Distributed Computing.* C++Now, 2013.

[2] Vijay Saraswat et al., *The Asynchronous Partitioned Global Address Space Model.* The First Workshop on Advances in Message Passing, 2011.

[3] Georgel Calin et al., *A Theory of Partitioned Global Address Spaces.* http://arxiv.org/abs/1307.6590, 2013.

[4] Bradford L. Chamberlain, *A Brief Overview of Chapel.* 2013.

[5] hartmut Kaiser et al., *ParalleX: An Advanced Parallel Execution Model for Scaling-Impaired Applications.* International Conference on Parallel Processing Workshops, 2009.

Techniques de programmation sans verrous

Les verrous (mutex, sémaphores, etc...) sont des moyens de prévenir des conditions de course dans un programme parallèle. Les verrous sont un des moyens les plus populaires pour raisonner sur un programme parallèle. Les verrous sont aussi une des choses qui retiennent l'avancement de la programmation parallèle. Dans cette présentation, j'aborderai diverses techniques de programmation sans verrous tout en exposant les implications profondes d'avoir un modèle mémoire dans un langage de programmation.

Références LockFree

[1] Herb Sutter, *atomic<> Weapons*. C++ and Beyond, 2012.

[2] Jeff Preshing, *An Introduction to Lock-Free Programming*. Preshing on Programming, 2012.

[3] Tony van Eerd, *The Basics of Lock-Free Programming*. Boostcon, 2013.

[4] *Lockless Programming Considerations for Xbox 360 and Microsoft Windows*. Microsoft, 2013.

Mémoire transactionnelle logicielle

Dans le monde de la programmation parallèle, les verrous sont souvent un mal nécessaire. En effet, comment peut-on s'assurer de l'atomicité de certaines opérations sans eux? Jusqu'à tout récemment, aucune réponse satisfaisante n'avait été formulée. Ceci à par contre changé dans les dernières années avec tout l'effort mis dans la mémoire transactionnelle logicielle. Cette présentation expliquera le pourquoi et le comment de cette technique de programmation parallèle. De plus, on y discutera aussi de l'aspect plus pratique de la chose en abordant des thèmes comme l'intégration de la mémoire transactionnelle dans des langages de programmation et les toutes dernières générations de processeurs.

Références STM

[1] Simon Peyton-Jones & Tim Harris, *Programming in the Age of Concurrency: Software Transactional Memory.* Channel 9, 2006.

[2] Dave Boutcher, *Software Transactional Memory in GCC 4.7.* Linux.conf.au, 2013.

[3] Michael Neuling, *What's the deal with Hardware Transactional Memory!?!.* Linux.conf.au, 2014.

[4] Victor Luchangco et al., *Standard Wording for Transactional Memory Support in C++.* C++ Standards Committee Papers, 2014.