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 les moyens les plus souvent utilisés pour raisonner sur un programme parallèle. Ils ne sont par contre pas toujours les moyens les plus efficaces pour implémenter des algorithmes et des structures de données parallèles. Dans cette présentation, je ferais un bref survol de techniques permettant de se libérer des verrous dans un contexte multiprogrammé ainsi que des dangers qui guettent ceux qui voudrait se risquer à les utiliser.

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] Bruce Dawson *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 risque par contre de changer dans les prochaines 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.

[5] Simon Peyton Jones, *Beautiful Concurrency*. Beautiful Code, O,Reilly, 2007.

Calcul approximatif et processeurs neuronaux

Au cours des dernières années, le monde de l'informatique a progressivement amorcer un virage vers la programmation parallèle. Les gains les plus souvent cités de ce virage sont ceux relevant de la performance des applications multiprogrammées. Cependant, il ne faudrait pas oublier que des gains ont aussi été faits au niveau de la consommation d'énergie. Dans cette présentation, j'introduirai un nouveau paradigme de programmation, le calcul approximatif, qui pousse encore plus loin cette idée d'économie d'énergie. De plus, je discuterai aussi d'un nouvel accélérateur basé sur les réseau de neurones développé pour supporter ce paradigme.

Références NPU

[1] Hadi Esmaeilzadeh et al., *Neural Acceleration for General-Purpose Approximate Programs*. Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, 2012.

[2] Hadi Esmaeilzadeh et al., *Neural Acceleration for General-Purpose Approximate Programs*. Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, 2012.

[3] Jie Han & Michael Orshansky, *Approximate computing: An emerging paradigm for energy-efficient design*. Proceedings of the 18th IEEE European Test Symposium, 2013.

Détection de conditions de course

Il va sans dire, corriger un programme parallèle est une des tâches les plus ardues auxquelles un programmeur peut être confronté. Dans l'espoir d'aider les programmeurs à régler des problèmes de multiprogrammation, plusieurs outils de détection de conditions de course ont été développé. Cette présentation se voudra une introduction à la théorie et aux algorithmes traînant derrière de tels outils.

Références TSan

[1] Stefan Savage et al., *Eraser: A Dynamic Data Race Detector for Multi-threaded Programs*. ACM Trans. Comput. Syst., Nov. 1997

[2] John Erickson et al., *Effective Data-race Detection for the Kernel*. Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation, 2010

[3] Konstantin Serebryany & Timur Iskhodzhanov, *ThreadSanitizer: Data Race Detection in Practice*. Proceedings of the Workshop on Binary Instrumentation and Applications, 2009

[4] Utpal Banerjee et al., *A Theory of Data Race Detection*. Proceedings of the 2006 Workshop on Parallel and Distributed Systems: Testing and Debugging, 2006

Algorithmes insensibles à la cache

Un bon algorithme devrait être conscient du fait qu'il sera exécuté sur une machine possédant une hiérarchie mémoire comportant souvent plusieurs paliers. Il ne devrait cependant pas être trop attaché à la hiérarchie mémoire d'un processeur donné pour ne pas perdre de sa portabilité. Dans cette présentation, j'explorerai cette idée d'algorithmes conscients sans trop être attachés à la cache dans le contexte de systèmes concurrents et parallèles.

Références cache

[1] Erik Demaine, *Programming in the Age of Concurrency: Software Transactional Memory*. Lecture notes from the EEF summer school on massive data sets, 2002.

[2] Harald Prokop, *Cache-Oblivious Algorithms*. Master's thesis, 1999.