Synthèse Workshop ORAP
«Evolution des processeurs: que faut-il anticiper pour les codes ? »

Résumé

Il est encore trop tôt pour définir précisément l'architecture logiciel du code du futur qui n'aura pas d'autre choix que de s'appuyer sur des architectures matérielles opérationnelles. Il apparaît que très probablement seules des solutions hétérogènes permettront d'atteindre l'exascale, ce qui aura pour conséquence la prise en compte de plusieurs niveaux de parallélisme. Cet accroissement de la complexité aura un impact sur l'environnement des développements. Ce document essaye de lister les divers points à prendre en compte pour anticiper les changements à venir.

Préambule

Ce document rapporte la synthèse des discussions sur le thème «Evolution des processeurs, que faut-il anticiper ? » dans le cadre d'un workshop organiser par Orap**.** Ce workshop a eu lieu le 5 décembre 2013, chez Total à La Défense. L'objectif de ce workshop était de faire un point d'étape sur les conséquences de l'évolution technologique sur les développements des codes scientifiques. Les besoins annexes (Big Data, visualisation, ….) même si ils sont importants ne sont pas traités ici.

Contexte

La stagnation de la fréquence des processeurs conduit naturellement vers une augmentation du nombre de cœurs de calcul. Conséquence de cette évolution, le parallélisme multi-coeur doit être pris en compte dans le cadre de l'évolution des applications existantes ( ) et considéré comme un point central de la conception de toutes nouvelles applications HPC. De plus, alors que nous assistons à l'accroissement de la largeur des instructions vectorielles, nous constatons que le débit mémoire reste(ra) (très) insuffisant pour alimenter les unités de calcul : nous sommes maintenant TRES loin des 24B/Flop des CRAY 1 ! En outre la localité des accès aux données, déjà un enjeu majeur, devient critique.

Le Tableau 1 indique les caractéristiques d'architectures contemporaines et leurs relations avec un objectif exaflopique (supposé en 2020) comme illustration de la trajectoire des systèmes HPC. Bien que cela représente un scénario extrême dans l'évolution, cela indique les ordres de grandeurs d'évolution du parallélisme pour des codes qui devront rester efficace dans les 10 à 20 ans qui viennent. Pour quelques cas particuliers, les technologies

exaflopiques doivent être envisagés (« ») mais il est probable que l'usage efficace de ces technologies se fasse surtout sur la base de « ». Cependant, dans tous les cas, au niveau des nœuds de calcul il sera nécessaire d'exhiber un parallélisme massif.

	Typique (CPU)	Valeur exemple (Xeon Phi)	Valeur exemple (GPU)
Fréquence	**1 – 3.2 GHz**	**1 GHz**	**.7-1.5 GHz**
Cœurs par processeur	**8-32**	**61**	**500-900**
Noeuds	**10k-100k+**	**1K-50k**	**1k-20k**
Peak Flops	**20.1P (Sequoia)**	**48P(+6.9P) (Tianhe-2)**	**24.5P(+2.6P) (Titan)**
Linpack Flops	**17.2P**	**33.9P**	**17.6P**
Facteur multiplicatif pour atteindre l'exaflop (10^{18})	**x 58**	**x 29.5**	**x 56.8**

Tableau 1 : Ordre de grandeur des machines actuelles (P for Petaflops)

L'évolution des technologies de processeurs impacte de multiples facteurs de décision dans la conception et la maintenance des codes. L'évolution des codes « legacy » dans ce contexte pose un challenge particulièrement difficile. En effet, nombre de ces codes « legacy » comportent plus d'1 million de lignes de code accumulées pendant des décennies et où cohabitent de nombreuses strates linguistiques ([F66], F77, F90, cohabitation C/C++ + F90, …). Par ailleurs, la plupart ont été pensés en séquentiel. « Dans une vielle bâtisse, en voulant remplacer un carreau de carrelage, c'est souvent l'ensemble du mur qui vient», cette métaphore [TG] illustre malheureusement de nombreuses situations existantes où la modularité du code a souvent été mise à mal par les ajouts successifs de fonctionnalités successifs. Par ailleurs, la remise en cause d'une partie de ces codes induit des problèmes de validation parfois insolubles et freine naturellement la transformation en profondeur d'un code, qui serait parfois nécessaire pour une mise en œuvre parallèle efficace. De plus, l'absence de visibilité sur les standards de programmation et technologies matérielles gêne considérablement la planification des mutations.

Il est important de noter que le premier facteur d'évolution d'une application reste son écosystème ainsi que son déploiement vers les utilisateurs. Un investissement pour franchir le palier technologique du parallélisme massif est difficile à justifier à court terme par rapport à un palier fonctionnel. Cependant, il est certain que le manque d'anticipation va conduire de nombreux codes vers la tombe !

La plupart des codes parallèles sont mis en œuvre via MPI où 1 cœur / 1 processus a été le meilleur compromis pendant des années. Cette stratégie est à bout de souffle dans de nombreux cas, incapable de tirer parti de la structure très hiérarchique des systèmes mémoire actuels. Les codes fondés sur le mix MPI + OpenMP sont rares mais en nombre croissant.

Considérations importantes pour le développement des codes

Efficacité des architectures : On constate avec les architectures modernes une perte continue de l'efficacité des mises en œuvre. Par exemple sur Linpack/Top500 (cas éminemment favorable) l'efficacité varie grandement avec l'hétérogénéité des composants des nœuds comme illustré dans le Tableau 2.

Machine	Efficacité Top500	Machine	Efficacité Top500
K Computer (Sparc)	93.17%	Titan (CPU+GPU)	64.88%
SuperMUC (x86_64)	90.96%	Tianhe-2 (CPU+Phi)	61.68%
Sequoia (BG/Q)	85.3%	PizDaint (CPU+GPU)	80.51%
Pleiades (X86_64)	73.15%	NCAR (X86_64)	83.64%
Archer (X86_64)	83.10%		

premières machines du Top 500 de novembre 2013, on obtient une puissance crête de 132 Petaflops pour 65 Mw, ce qui conduirait par une simple règle de trois à une machine exaflopique d'une consommation de presque 500 Mw.

OpenMP : L'usage d'OpenMP avec une parallélisation à grain fin est une approche majoritairement adoptée. Elle est simple à déployer mais produit des codes moins
que les approches à gros grain qui sont malheureusement plus complexes à mettre en œuvre surtout dans les codes « legacy .

Survivre à des générations successives de machine : La durée de vie typique d'une machine est de l'ordre de 5 ans et celle des applications de 30 ans. Les principaux codes «subiront» 6 machines probablement très différentes entre leur début et leur fin de vie. Ce chiffre est un minimum : en effet les grands codes sont appelés à être utilisés sur des moyens externes à l'organisme ou à l'entreprise qui en est le maître d'œuvre. Seuls les standards de programmation/technologies pérennes permettent à ces codes de survivre. Il est important de noter que cette durée de vie implique des déploiements multi-OS et multi-chaînes de compilation. L'histoire montre que les passages successifs d'une technologie parallèle (e.g. Fortran vectoriel à Fortran + OpenMP puis à Fortran + MPI) à une autre se fait de plus en plus difficilement.

La programmation homogène tire à sa fin : Il faut se préparer à vivre dans un monde hétérogène composé de plusieurs niveaux de parallélisme : Macro (e.g. MPI), Méso (e.g. parallélisme entre threads), Micro (e.g. parallélisme de type SIMD, parallélisme de données).

Instructions vectorielles : L'impact négatif de la non-vectorisation (e.g. génération des instructions AVX) des codes ne peut être négligé (typiquement entre 50% et 85% suivant la micro-architecture). L'accroissement de la largeur des instructions vectorielles tend à accroître cette pénalité.

Bibliothèques : L'usage des bibliothèques est un élément majeur dans la simplification du développement des codes. L'impact des bibliothèques sur la portabilité des codes est déterminant. Leur choix ne doit pas être dicté uniquement par des éléments de performances mais aussi par des éléments de stabilité et de disponibilité sur les systèmes.

Organisation des données : L'organisation des données doit répondre à de nombreuses contraintes : localité des accès au données, entrées/sorties, accès asynchrones pour le recouvrement des calculs avec les entrées/sorties, intégration des pré- et post-traitements, limitation du volume pour la mise en œuvre de « checkpoint-restart », etc. Une approche holistique de ces considérations est cruciale compte tenu de l'évolution des hiérarchies mémoire et de la difficulté à transférer de grand volume de données (relativement aux capacités de calculs des systèmes).

Augmentation du coût des entrées/sorties : Très souvent négligé, en raison du peu d'attrait du sujet (difficile et austère), le coût des entrées/sorties augmente et va devenir dominant sur les futurs systèmes. La séparation entre le cœur de calcul de la simulation et les pré- et post-traitements est vouée à l'obsolescence ; Intégrer certains pré- et post-traitements au cœur de calcul du code va devenir de plus en plus nécessaire (par exemple. visualisation in-situ évitant de produire et reprendre à posteriori de grandes masses de données de résultats de calcul).

Prise en compte des configurations de machines : Nous entrons dans l'ère des intelligents offrant des possibilités d'adaptation aux différentes configurations matérielles. Bien que des propositions au sein de la communauté recherche soient disponibles et en capacité à répondre à de nombreux problèmes, il manque encore des efforts de standardisation.

Processus de développement : Le cycle de développement en V a montré ses limites. Le développement des applications HPC comporte peu de spécificités du point de vue de l'ingénierie du logiciel sinon un besoin important autour de l'optimisation des codes. Les technologies (e.g. Hudson) et méthodologies de développement agiles (e.g. SCRUM) peuvent apporter des bonnes pratiques à la communauté HPC. Il est à noter que les économies sur les parties traditionnellement non «nobles» (maintenance, parties hors calcul) n'en sont qu'à très court terme. La complexité croissante induite par l'exascale conduira à accroître la taille des équipes de développement qui seront multidisciplinaires (physique/numérique/informatique), l'utilisation de bonnes pratiques de développement logiciel comme la gestion de configuration est un prérequis.

Scalabilité : Avec l'accroissement important du nombre de cœurs, de nombreux code (surtout MPI) ne seront pas . La résolution de ce problème peut avoir des conséquences importantes allant jusqu'à la remise en cause du schéma numérique (y compris la discrétisation du problème). Voir la section suivante.

Technologie HPC : La Communauté HPC est restreinte et manque de masse critique (que ce soit offre ou demande) ce qui limite l'offre technologique (en particulier logiciel).

Réunir les compétences : L'évolution des codes oblige à réunir des compétences plus larges que celles traditionnellement réunies dans les projets HPC. La complexité d'exploitation du parallélisme massif hybride ne peut se faire par la seule formation des scientifiques « applicatifs » à l'ensemble des technologies logicielles, matérielles, algorithmiques mises en jeux. L'organisation des compétences scientifiques, numériques et informatiques est un challenge organisationnel qui doit trouver une expression compatible avec chaque écosystème. Les architectures logicielles des projets de développement sont ici cruciales pour permettre des interactions constructives entre les communautés.

Les compromis algorithmiques

Il est probable que l'évolution des architectures pousse à revenir aux équations de la physique plutôt que de faire évoluer les codes aux forceps. Le compromis issu des décennies précédentes favorisant la diminution des volumes de calcul au détriment de la régularité de ceux ci ne correspond plus toujours à une solution efficace. Le coût des flops est en décroissance alors que celui de l'octet et du déplacement de données est en forte croissance. Il faut donc très probablement envisager de revenir à la feuille blanche pour repartir sur des bases plus adaptées aux équilibres des systèmes futurs et d'assurer une scalabilité forte des codes. Comme l'illustre la Figure 1, la parallélisation, les méthodes numérique et la physique sont imbriquées, ces premières guidant ce qui peut être simulé.

« » sont une piste qui doit être suivie et amplifiée. Ces considérations doivent être évaluées à l'aune de ce qui est réaliste et économiquement faisable. Pour les « » codes, les approches incrémentales sont préférable lorsque techniquement possibles.

Recommandations et bonnes pratiques

Cette section recense quelques recommandations et bonnes pratiques à considérer pour la migration d'un code « ou la conception d'un nouveau code de simulation. Cependant, il est fondamental de les inscrire dans le contexte de développement afin que celles-ci soient en adéquation avec les moyens, les utilisateurs et technologies disponibles.

Validation : La validation doit être un processus continu au sens de l'ingénierie du logiciel (technique d'intégration continue par exemple). On doit s'assurer de la reproductibilité des résultats sans toutefois dépendre de comparaisons bit à bit mises à mal par les exécutions parallèles (nombre variable de OpenMP utilisés d'un à l'autre, variation de l'ordre d'arrivée des messages, …).
La consistance de l'utilisation et la vérification visuelle des données sont essentielles pour la confiance dans les résultats. Cet aspect doit faire partie intégrante de la migration d'un code.

Données : Localité, affinité et volume des données sont des considérations critiques. Les structures internes de codes doivent pouvoir évoluer facilement pour s'adapter à l'architecture cible. Par exemple prévoir que des structures en tableau pourront être converties en tableaux de structures (et vice versa). Cette considération tend à orienter les mises en œuvre vers C++ et l'usage des (mais ce n'est pas l'avis de tout le monde).

Scalabilité : La mise en œuvre doit éviter autant que faire se peut les communications collectives en préférant absolument les échanges de données entre voisins. L'usage de solveurs minimisant les communications est à préférer. Bien entendu ces considérations sont des conséquences des décisions amont dans la structure de données et les schémas numériques.

Langage de programmation : Le choix du langage Fortran, C++, etc. doit d'abord être fait suivant les compétences des développeurs. Fortran reste un choix pertinent dans de nombreux cas. Dans un certain nombre de cas on trouve l'usage de C++ pour la gestion des données et du d'exécution, Fortran est alors utilisé pour les calculs.
Sur les machines HPC actuelles, les seuls langages "universellement supportés" sont le C, C++, et Fortran (1995 et une bonne partie de 2003, il y a encore peu de 2008). D'autres langages prometteurs manquent encore de "masse critique" pour que des codes à vie longue puissent les envisager sérieusement, même s'ils paraissent attrayants.

Les langages peuvent être combinés, mais la gestion de l'interopérabilité nécessitant un peu de code additionnel, les frontières sont idéalement sur des interfaces internes relativement stables dans le temps. Le fait de s'appuyer sur des équipes spécialisées conduira obligatoirement à des codes multi-langages, dont malgré tout le nombre devra rester restreint pour des raisons de portabilité et de pérennité.

Les avantages et inconvénients de chacun se situent au niveau d'une multitude de détails, et l'intégration de bibliothèques externes, le lien avec les pré et post-traitement, etc. peuvent à terme faire pencher la balance différemment.

Approches spécifiques à un domaine : Les approches fondées sur une programmation spécifique au domaine (DSL) sont de plus en plus populaires. Elles fournissent un niveau d'abstraction important aux utilisateurs/programmeurs scientifiques tout en autorisant la mise en œuvre de techniques informatiques sophistiquées. Ces approches servent de pont entre les communautés. Attention cependant à la rupture méthodologique que cela peut entraîner.

Processus de développement : Le processus de développement doit mettre en œuvre les méthodes et techniques de l'ingénierie du logiciel tel que :

- Contrôle par des outils automatiques (+ scripts automatiques) ;
- Gestion de versions ;
- Mesures de performance intégrées au code ;
- Tests unitaires ;
- Bases de non-régression.

Les coûts induits par les erreurs détectées tardivement et les déstructurations des codes par des pratiques approximatives sont très importants. Il ne s'agit pas d'une nouveauté mais la mise en œuvre de parallélisme massif et hiérarchique complexifie les applications de manière suffisante à rendre les processus de développement ancien obsolète.

Tolérance aux fautes / pannes: La tolérance aux pannes telle qu'envisagée sur les systèmes exaflopiques n'est pas à l'ordre du jour : cependant l'augmentation du coût des entrées/sorties oblige à des stratégies minimisant les volumes de données nécessaires au . Les approches applicatives (plutôt que système) sont actuellement les seules à même d'être à terme.

Entrées / sorties (I/O): Pour faire face au coût des entrées/sorties, le recouvrement avec les calculs est souvent une nécessité. Par exemple, on peut envisager que des cœurs de calcul puissent être dédiés aux I/O afin d'avoir une mise en œuvre asynchrone des transferts des données des codes de calcul vers les serveurs. En particulier il s'agit d'obtenir un lissage des pics I/O tout au long de la simulation. Cette considération dépend de l'architecture, de l'organisation des données et des volumes des différentes mémoires.

Intégration des pré- et post-traitements : La manière dont ils sont couplés avec la partie calcul doit être finement analysée afin de minimiser les stockages de données. L'usage des données doit être bien défini/compris (au niveau de l'ensemble de la chaîne de calcul/analyse) pour apporter une solution viable à ce problème.

Architectures des codes : L'architecture des codes doit être maîtrisée et respectée (« enforced »). Les défauts de l'architecture et sa dissolution dans le temps ont des conséquences graves. Les objectifs de l'architecture sont :

- Amélioration de la modularité ;
- Faciliter la réécriture de certains sous-ensembles de fonctionnalités ;
- Faciliter l'intégration des compétences et évolution des technologies ;
- Fournir des API externes et surtout INTERNES au code ;
- Permettre la présence de versions ultra optimisées (performance sur UNE architecture) aux cotés de versions de références (pérennité du code). Attention à l'illusion que pourrait procurer des approches de haut niveau. Certaines parties du code devront pouvoir « coller » à la machine sans pour autant déstructurer l'architecture ;
- Dans la mesure du possible le solveur doit être intégré de manière « plug-and-play » afin de pouvoir choisir un algorithme efficace sur l'architecture cible. Cette considération va généralement au-delà de l'aspect API.

Règles de mise en œuvre : Il s'agit de définir des règles et structures de codes pour guider les développeurs. Ces règles sont spécifiques à chaque application et à son écosystème. Réduire les coûts passe par la mise en œuvre d'un certain nombre de règles (une ligne de code deboguée et documentée coûte entre 10 et 100 € suivant la complexité de l'algorithme) qui doivent être mises en pratique. La définition de règles inclut la proposition de structures de code typiques pour aider à la bonne vectorisation etc.

Bibliothèques : Les bibliothèques apportent des mises en œuvre optimisées des procédures fréquentes. L'utilisation de bibliothèques externes doit être mesurée avec précaution tant pour des motifs de pérennité et de disponibilités sur des systèmes variés que sur la complexité du de l'applications. Les règles de bonne conduite ici sont :

- Utiliser au maximum des bibliothèques constructeur (ou) optimisées,
- Ne pas utiliser de vieux algorithmes qui ont été pensés séquentiellement (e.g.),
- Choisir des bibliothèques pérennes ou facilement remplaçables afin de limiter l'adhérence au système courant.

Runtime : Les fournissent les services intermédiaires de gestion des ressources de la plateforme qui ne sont pas fournis en natif par le système d'exploitation (e.g. et). Le recours à des performants proposant un robuste offre des capacités d'adaptation à la configuration de l'architecture cible. Compte tenu du nombre de à gérer dans le futur, l'usage d'ordonnanceur hiérarchique moins précis sera

nécessaire avec comme conséquence une dégradation de la capacité à mettre en œuvre un équilibrage de charge fin.

Deboggage : Le coût des augmente avec le délai de détection. L'usage d'outils d'analyse de code tel que est une pratique importante malgré le surcoût à l'exécution (x10).

Vecteur / parallélisme de données : Les calculs vectoriels et/ou data parallèles à grain fin (e.g. orientés GPU) contribuent pour une part très significative (e.g). La mise en œuvre ne peut être laissée au seul compilateur, le code doit être écrit afin de grandement faciliter (règles / structures de codes) la production de la vectorisation par le compilateur (e.g. directives). L'usage des intrinsèques [2] est à déconseiller car peu lisibles et peu portables.

Veille technologique : Anticiper a un coût, d'autant plus que les pistes sont multiples. Éviter la tentation de tout ce qui est nouveau, mais agir à temps, sans procrastination... La veille technologique est déterminante pour faire les bons choix.

Réunir les compétences : Des initiatives nationales et internationales telles la « Maison de la simulation » (http://www.maisondelasimulation.fr), les centres de compétences, etc. devraient être sollicitées.

Organisateurs Membres du Conseil Scientifique de l'ORAP

- Bodin François, Université de Rennes 1
- Calandra Henri, Total
- Refloc'h Alain, Onera

Participants

- Alimi Jean-Michel, Observatoire de Paris
- Colin de Verdière Guillaume, CEA DAM
- Courteille François, Nvidia
- Dinh Quang, Dassault Aviation
- Dolbeau Romain, CAPS entreprise
- Fournier Yvan, EDF
- Grigori Laura, Inria Rocquencourt
- Guignon Thomas, IFPEN
- Michel Kern, Inria (contribution additionnelle hors ligne)
- Meurdesoif Yann, CEA
- Namyst Raymond, Inria Bordeaux
- Petiton Serge, Université de Lille 1, Sciences et Technologies
- Ricoux Philippe, Total
- Thierry Philippe, Intel

[2] Fonctions codées en assembleur qui peuvent être insérées dans un programme.