Modeling performance of serial and parallel sections of multi-threaded programs in manycore era

Surya Narayanan Natarajan

ANNÉE 2015

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

École doctorale Matisse

présentée par

Surya Narayanan KHIZAKANCHERY NATARAJAN

préparée à l'unité de recherche INRIA – Bretagne Atlantique
Institut National de Recherche en Informatique et Automatique
Composante Universitaire (ISTIC)

Modeling performance of serial and parallel sections of multi-threaded programs in many-core era

Thèse soutenue à Rennes le 01, Juin, 2015

“You can never be overdressed or overeducated.”
- Oscar Wilde

Acknowledgements

First of all, I would like to thank the jury members PAZAT Jean-Louis (President of the Jury), GOOSSENS Bernard and GOGLIN Brice (Reviewers of the thesis), HEY-DEMANN Karine and COLLANGE Sylvain (Examiners of the thesis) for giving me this opportunity to defend my thesis.

SEZNEC Andre, my thesis adviser told me a wise statement once that *"Doing a PhD is not about mastering a subject but to learn the art of doing an extensive research"* which was an eye-opener for me to take up the assigned topic under him. After, being his student for 3 years, I truly believe in his statement and thank him for being a great mentor who gave me all the freedom and timely support to lift me out of my deadlocks.

I would like to thank my parents who always wanted me to excel in my academics[1]. Without their love, sacrifice and support I would not have come so far in my life.

I thank my wife Shruti for being a moral support especially at the end of my thesis when I was not able to visit her often. I hope I will compensate this by living with you for the rest of my life.

I am grateful to the SED team of INRIA who helped me in solving every issue I had by setting up a new experimental platform for me and also to my colleague PERAIS Arthur who helped me in writing my french resume.

Last but not the least, I would like to raise a virtual toast to all my friends who made my life easy, comfortable and enjoyable all these 3 years in Rennes. Hope to see you again frequently and have those long philosophical conversations. CHEERS!!!

[1]Clichéd Indian parents :P

Contents

Résumé en Français

Contexte

Ce travail a été effectué dans le contexte d'un projet financé par l'ERC, **Defying Amdahl's Law** (DAL) [Sez10], dont l'objectif est d'explorer les techniques micro-architecturales améliorant la performance des processeurs *many-cœurs* futurs. Le projet prévoit que malgré les efforts investis dans le développement de programmes parallèles, la majorité des codes possèderont toujours une quantité signifiante de code séquentiel. Pour cette raison, il reste primordial de continuer à améliorer la performance des sections séquentielles des-dits programmes. Au sein de cette thèse, le travail de recherche se concentre principalement sur l'étude des différences entre les sections parallèles et les sections séquentielles de programmes *multithreadés*[2] (MT) existants. L'exploration de l'espace de conception des futurs processeurs many-cœurs est aussi traitée, tout en gardant à l'esprit les exigences concernant ces deux types de sections ainsi que le compromis performance-surface.

De l'unicœur au multicœurs, puis au many-cœurs

L'industrie des processeurs est passée des processeurs simple cœur aux processeurs *multicœurs* [PDG06] il y a presque une décénnie afin de mitiger certains problèmes tels que la difficulté d'augmenter la fréquence d'horloge ou la forte dissipation thermique. Ces problèmes – combinés à l'augmentation exponentielle de la complexité des processeurs faisant que la performance n'augmentait que de façon incrémentale – ont mené à l'avènement de l'ère des multicœurs. Conséquemment, les efforts de l'industrie se sont portés sur l'augmentation du nombre de cœurs dans un processeur possédant une architecture moins complexe mais plus efficace d'un point de vue énergétique. Cependant, de nouveaux problèmes dûs aux multicœurs ont émergés, principalement liés à l'utilisation de ressources partagées (mémoire cache, mémoire centrale, réseau d'interconnexion) par les programmes *multithreadés*. Ces nouvelles difficultés ont encouragé les chercheurs à étudier les architectures parallèles et en particulier des solutions pour permettre l'exécution efficace des sections parallèles des programmes sur ces architectures.

[2]Un *thread* correspond à un flot de contrôle exécuté sur un *cœur* (logique ou physique) d'un processeur.

Initialement, les multicœurs étaient utilisés uniquement par les scientifiques pour le calcul scientifique ainsi que par certaines industries requérant de hautes performances de calcul. Cependant, aujourd'hui, les multicœurs sont présents dans de nombreux appareils tels que les ordinateurs portables et les stations de travail. De plus, les appareils embarqués tels que les téléphones portables et les tablettes requièrent une consommation faible tout en fournissant une puissance de calcul élevée, privilégiant l'utilisation de multicœurs. Suivant cette tendance, l'industrie et l'académie ont déjà commencé à étudier la suite logique aux processeurs multicœurs, les processeurs many-cœurs.

"Many-cœurs" ou "Kilo-cœurs" est un terme à la mode depuis quelques années. Il s'agit en fait d'une seule puce (processeur) possédant des centaines de cœurs. Ces puces pourraient être accessible dans seulement quelques années. Quand des processeurs à 4 ou 8 cœurs sont essentiellement utilisés pour exécuter plusieurs programmes distincts en parallèle, les many-cœurs possédant ces centaines de cœurs requerront des programmes hautement parallèles pour tirer parti de leur puissance de calcul potentielle. Les many-cœurs seront utilisés soit pour réduire le temps d'exécution d'une application donnée sur un certain jeu de données (i.e. afin de réduire le temps de réponse), soit pour permettre l'augmentation de la taille du problème à traiter dans un temps donné (i.e. fournir un meilleur service).

Défis existants à l'heure des many-coeurs

A court terme, la loi de Moore [Moo98] continuera à fournir aux architectes davantage de transistors par unité de surface, c'est à dire plus de cœurs si l'on considère la tendance actuelle. Le défi principal étant d'utiliser ces transistors afin d'obtenir un niveau de performance correspondant aux ressources supplémentaires disponibles, tout en gardant à l'esprit les contraintes énergétiques dues à l'activation de tant de transistors sur une surface si faible. La performance des multicœurs provient principalement de la division des calculs entre plusieurs *threads* s'exécutant en parallèle. Cependant, cette performance peut être limitée par de nombreux goulets d'étranglement logiciels ou matériels. Du point de vue logiciel, le modèle de programmation utilisé pour paralléliser le programme peut être inefficace ou le programme peut n'avoir pas été parallélisé de manière à utiliser la puissance de calcul disponible e.g. : sections critiques [EE10], partage de ressources partagées non nécessaires, etc. Du point de vue matériel, l'accès à une ressource architecturale partagée par plusieurs *threads* peut créer de la contention sur cette ressource [GKS12] e.g. : bande passante mémoire, caches partagés, sur-utilisation du protocole de cohérence de cache, etc. De ce fait, la performance des programmes parallèles dépend de l'architecture matérielle sous-jacente et du modèle de programmation utilisé pour paralléliser le programme.

Tout les défis mentionnés plus haut se concentrent sur la conception de *kernels* parallèles au sein d'un unique programme. Lorsque le programme entier est considéré, différents problèmes fondamentaux doivent être résolus. En effet, les many-cœurs actuels sont plutôt considérés comme des accélérateurs augmentant la performance de programmes hautement parallèles. Cependant, même dans ce cas, certaines sections du programme ne peuvent être parallélisées ou ne le sont pas par ignorance d'une telle possibilité. Ces sections séquentielles du programmes ne peuvent être négligées.

Modèles de la performance parallèle actuels

Deux modèles, la loi d'Amdahl [Amd67] et la loi de Gustafson [Gus88], sont encore communément utilisés pour extrapoler la performance théorique des programmes parallèles. Ces deux lois ne sont cependant que deux règles approximatives. La loi d'Amdahl calcule l'augmentation de performance théorique d'un programme en considérant une quantité fixe de travail réalisée par un nombre cœurs variable. La loi de Gustafson détermine l'augmentation de performance en terme de travail réalisé. Cette loi considère que la partie parallèle d'un programme grandit linéairement avec le nombre de cœurs P – comme illustré par la Figure 1 – alors que la partie séquentielle reste constante. Ces dernières années, des variations de ces deux lois ont vu le jour afin de prendre en compte le nombre de cœurs grandissant des multi et many-cœurs. La majorité des travaux sur la performance s'est concentrée sur la dégradation de la performance arrivant uniquement lors de l'éxécution des sections parallèles et dûes à la synchronisation et la communication entre les différents *threads* [EE10], [JM12], [YMG14]. Ces travaux n'ont pas étudié l'impact des sections séquentielles sur la performance des programmes parallèles. Cette thèse étudie les goulets d'étranglement matériels et logiciels existants à cause de sections séquentielles explicites qui sont restés ignorés malgré l'avènement des multicœurs, mais deviennent de plus en plus proéminents et peuvent limiter le passage à l'échelle à l'ère des many-cœurs.

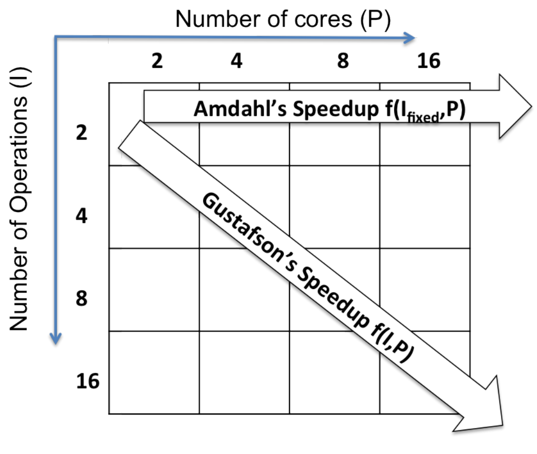


Figure 1: La loi d'Amdahl considère une quantité de travail fixe quand la loi de Gustafson considère que la quantité de travail passe à l'échelle avec le nombre de cœurs.

Directions de recherche

Les programmes parallèles sont composés de sections parallèles et séquentielles. Une section séquentielle représente une partie du programme où un unique *thread* est actif. En général, le rôle de ce *thread* est de contrôler l'exécution des *threads* parallèles et d'exécuter les sections qui ne peuvent être parallélisées. Au sein de cette thèse, le travail de recherche se concentre sur l'obtention de réponses aux questions suivantes :

1. *Au sein d'un programme parallèle, les sections parallèles sont-elles différentes des sections séquentielles?* Il est important de savoir si les sections séquentielles et parallèles ont des caractéristiques indépendantes de la micro-architecture différentes. Par exemple, leurs combinaisons d'instructions, empreinte mémoire, flot de contrôle (branchements), motifs d'accès mémoire, etc. Ces informations sur un programme ou un groupe de programmes sont primordiales pour la conception d'un many-cœur ayant un meilleur ratio performance/surface.

2. *Comment modéliser le passage à l'échelle d'un programme sur les many-cœurs futurs?* Les sections séquentielles et parallèles ne passeront pas forcément à l'échelle de la même manière, ce que les modèles analytiques existant ne prennent pas en compte. De ce fait, une méthode d'évaluation de la performance considérant deux facteurs de passage à l'échelle différent est requise afin de déterminer le comportement du programme lorsque le nombre de cœur augmente.

Contributions

Deux contributions sont décrites au sein de ce document.

- **Le modèle de performance "Serial/Parallel Scaling" (SPS)** Nous avons développé une méthodologie permettant de distinguer les temps d'exécution individuels des sections séquentielles et parallèles au sein d'un programme parallèle. Nous avons utilisé *tiptop* [Roh11], un outil de surveillance de la performance pouvant échantillonner et enregistrer différentes métriques décrivant l'exécution (e.g. le nombre de cycles ainsi que le nombre d'instructions exécutées) de différents *threads*. Les principaux avantages de *tiptop* sont 1) un faible coût d'échantillonage en terme de temps d'exécution, donc adapté à l'édude des many-cœurs. 2) *tiptop* ne requiert ni d'annoter le code source du programme parallèle, ni d'instrumenter le code à l'exécution. En utilisant les données mesurées, nous avons construit notre modèle "Serial/Parallel Scaling" (SPS) pour chaque application exécutée sur l'architecture que nous avons considérée. Le modèle SPS utilise la taille du jeu de données (I) et le nombre de cœurs (P) afin de déterminer le temps d'exécution d'un programme parallèle. Notre modèle empirique est donné dans l'Equation 1.

$$t(I,P) = c_{seq}I^{as}P^{bs} + c_{par}I^{ap}P^{bp} \tag{1}$$

Le modèle SPS utilise six paramètres obtenus empiriquement afin de représenter le temps d'exécution de programmes parallèles. Il prend en compte la taille du jeu de données ainsi que le nombre de cœurs. c_{seq}, as et bs sont utilisés pour modéliser le temps d'exécution de la partie séquentielle du programme. c_{par}, ap et bp sont utilisés pour modéliser le temps d'exécution de la partie parallèle du programme.[3] c_{seq} et c_{par} sont des constantes pour la partie séquentielle et la partie parallèle donnant une estimation initiale du temps d'exécution. as et ap sont les paramètres de passage à l'échelle séquentiel et parallèle relatifs au jeu de données (*Input Serial Scaling* [ISS] et *Input Parallel Scaling* [IPS]). bs et bp sont les paramètres de passage à l'échelle relatifs au processeur (*Processor Serial Scaling* [PSS] et *Processor Parallel Scaling* [PPS]). Dans la suite de ce document, nous ferons référence à as, ap, bs et bp par leurs acronymes respectifs ISS, IPS, PSS et PPS.

Règles d'analyses du modèle SPS

Nous présentons un ensemble de règles empiriques permettant d'analyser le passage à l'échelle de programmes parallèles depuis les paramètres SPS collectés en utilisant un jeu de données de petite taille ainsi qu'un nombre de cœurs peu élevé.

- L'augmentation maximale de la performance est limitée par PPS : PPS proche de -1 indique qu'une augmentation de performance quasi-linéaire est possible. PPS proche de 0.5 limite l'augmentation de la performance à la racine carrée du nombre de cœurs.
- La partie séquentielle peut passer à l'échelle avec la taille du jeu de données : Si le passage à l'échelle se fait avec le même facteur que le passage à l'échelle de la partie parallèle ($ISS \approx IPS$), l'augmentation de performance sera peu modifiée par la taille du jeu de données et atteindra rapidement son maximum.

Nous considérons 9 programmes de PARSEC, LONESTAR et PLAST pour notre étude. Les modèles de temps d'exécution pour ces 9 programmes sur un processeur Xeon-Phi sont obtenus en utilisant des paramètres I et P tels que $\{I \leq 16, P \leq 16\}$ et sont validés sur des mesures obtenues pour $\{I = 32, P \leq 240\}$. Nous pouvons observer dans la Figure 2 que l'erreur médiane résultante de la prédiction de la performance d'un programme exécutant 240 *threads* est comprise entre 3% et 20%, ce qui est acceptable lorsque qu'un niveau d'abstraction tel que celui de SPS est utilisé. En particulier, SPS permet d'obtenir la limite haute de la performance pouvant être obtenue en augmentant le nombre de cœurs et/ou la taille du jeu de données.

[3]Puisque le temps d'exécution par cœur de la partie parallèle inclut le temps passé dans la synchronisation et la communication entre cœurs, notre modèle pour la partie parallèle les prend en compte par défaut.

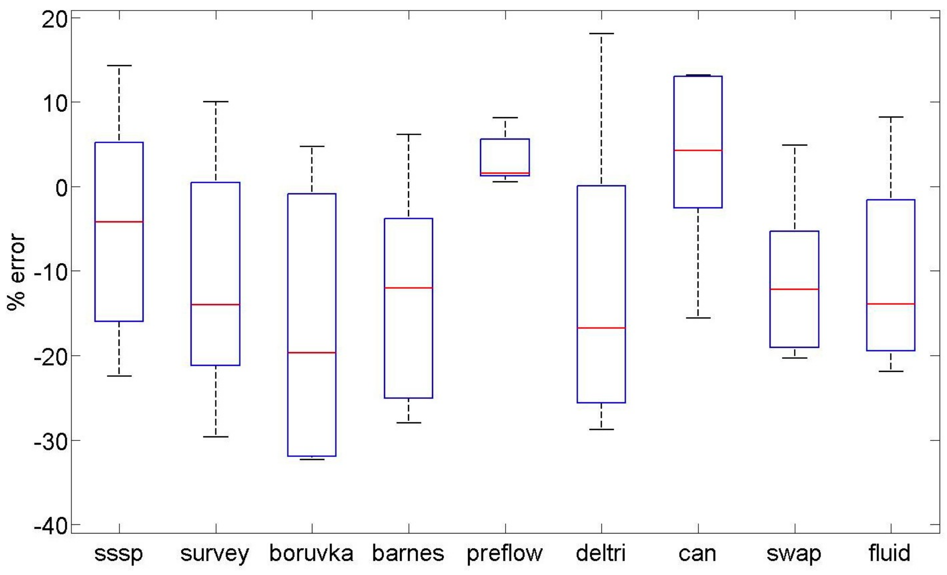


Figure 2: Diagramme en boîtes montrant les valeurs MIN, MEDIANE, MAX de l'erreur obtenue lors de la prédiction de la performance pour $\{I = 32, P \leq 240\}$, pour différentes applications.

Nous avons comparé l'augmentation de performance maximale des programmes parallèles considérés avec les valeurs prédites par les modèles analytiques existants (Amdahl [Amd67], Gustafson, [Gus88] et Juurlink et al. [JM12]) afin d'illustrer qu'en réalité, la partie séquentielle a un impact majeur sur le passage à l'échelle de certaines applications parallèles pour lesquelles la partie séquentielle grandit avec la taille du jeu de données. Notre modèle permet l'inférence de propriétés supplémentaires comme le fait que le poids des parties séquentielles et parallèles est dépendant de l'architecture. Conséquemment, les facteurs de passage à l'échelle desdites parties peuvent uniquement être déterminés en exécutant l'application afin de les mesurer explicitement. Finalement, dans le cas du passage à l'échelle de la partie séquentielle, utiliser un many-cœurs homogène peut être un choix inadapté, et nous avons donc montré la supériorité d'un many-cœurs hétérogène utilisant quelques cœurs complexes et très performant en conjonction d'un grand nombre de cœurs plus simples afin d'augmenter la performance du programme dans sa totalité.

- **Conception des many-cœurs du futur**

  Comme seconde contribution, nous avons analysé les caractéristiques intrinsèques des parties séquentielles et parallèles de programmes parallèles et découvert que ces caractéristiques diffèrent. La partie séquentielle est plus intensive au niveau de la mémoire et possède plus de branchements (flot de contrôle complexe), alors que la partie parallèle effectue plus de calculs et possède plus d'instructions vectorielles. Suite à ces observations, nous avons conduit une étude visant à définir les besoins matériels d'un cœur exécutant uniquement du code séquentiel ainsi que d'un cœur exécutant uniquement du code parallèle. Pour cela, nous avons utilisé des simulateurs à base de traces générées par un PINTOOL. Nous avons exploré trois types de ressources pour chaque type de cœurs : 1) Le nombre d'unités fonctionnelles requises. 2) Le prédicteur de branchement, puisque la partie séquentielle a plus de branchements difficile à prédire, et requiert donc un prédicteur de branchement complexe. 3) La taille du cache puisque la partie séquentielle a une empreinte mémoire plus grande que la partie parallèle, i.e. la localité des données est moins élevée et un cache privé de plus grande taille doit être implémenté. De ce fait, les many-cœurs possédant de nombreux cœurs simples afin d'exécuter la partie parallèle du programme ainsi qu'un faible nombre de cœurs complexes et performants afin d'exécuter la partie séquentielle du programme semblent plus appropriés pour l'exécution de programmes parallèles. Conséquemment, le many-cœurs du futur devrait être hétérogène.

Conclusion

A l'heure de many-cœurs sur lesquels de nombreux *threads* s'exécutent de façon plus ou moins concurrente, il est primoridal de comprendre d'où proviennent les goulets d'étranglement limitant le passage à l'échelle des programmes parallèles. Les modèles analytiques existants considèrent que la partie séquentielle est soit constante soit négligeable, et donc indépendante du nombre de cœurs et de la taille du jeu de données. Dans cette thèse, nous montrons que le passage à l'échelle de la partie séquentielle peut être un important facteur limitant pour certains programmes. La solution à ce problème est d'améliorer la performance de la partie séquentielle au maximum afin d'améliorer la performance du programme dans sa totalité. De plus, nous sommes convaincus que dans le futur, de nombreux programmes possèderont toujours du code purement séquentiel, que ce soit parce que ceux-ci ne sont pas parallélisables, ou parce que le programmeur n'aura pas sû exploiter le parallélisme existant. Cette partie séquentielle ne sera pas négligeable. De ce fait, le many-cœurs du futur sera hétérogène avec quelques cœurs complexes et performants combinés à de nombreux cœurs plus simples, le tout sur une unique puce.

Introduction

Context

This thesis work is done in the general context of the ERC funded **Defying Amdahl's Law** (DAL) [Sez10] project which aims at exploring the micro-architectural techniques that will enable high performance on future many-core processors. The project envisions that despite future huge investments in the development of parallel applications and porting it to the parallel architectures, most applications will still exhibit a significant amount of sequential code sections and, hence, we should still focus on improving the performance of the serial sections of the application. In this thesis, the research work primarily focuses on studying the difference between parallel and serial sections of the existing multi-threaded (MT) programs and exploring the design space with respect to the processor core requirement for the serial and parallel sections in future many-core with area-performance tradeoff as a primary goal.

Design focus in the processor industry has shifted from single core to multi-core [PDG06] almost a decade ago to combat issues like increasing clock frequency, high power consumption and high heat dissipation. These problems along with exponentially growing design complexity that resulted only in incremental performance improvement led to the multi-core era. Hence, the focus shifted towards increasing the number of cores in a processor which will target less complex, power efficient hardware architectures. However, with multi-core new issues came with respect to the utilization of shared resources like caches, memory and interconnects efficiently for the multi-threaded programs. This has changed the research direction towards parallel architectures and the focus more towards efficient execution of parallel parts of the program.

Initially, multi-core processors were used only by scientists for scientific computations and high performance computing industries. But, today, they have become omnipresent in every computing device like desktops and laptops. Moreover, the portable embedded devices like cellphones, tablets and wearable devices demands low power consumption to have longer battery life and high computation for performance that favors the utilization of multi-core processors. Following this trend, the industry and academia have already started thinking parallel and focusing on the so called many-core processors.

"Many-core" or "Kilo-core" has been a buzzword for a few years. Single silicon die featuring 100's of cores can be on-the-shelf in a very few years. While 4 or 8-cores are essentially used for running multi-program workloads, many cores featuring 100's of cores will necessitate parallel applications to deliver their best performance. Many-cores will be used either to reduce the execution time of a given application on a fixed working set (i.e to enable shorter response time) or to enlarge the problem size treated in a fixed response time (i.e., to provide better service).

For the foreseeable future, Moore's law [Moo98] will continue to grant computer architects more transistors per unit area of the silicon die or more cores with respect to the recent trend. The main challenge being how to utilize them to deliver performance and power characteristics that fits their intended purpose. Performance in multi-core processors are basically achieved by dividing computation among multiple threads running in parallel. Application performance can be limited by multiple bottlenecks from software and hardware side. From the software side, the programming model used to port the application might be inefficient or the application was not parallelized properly to exploit the available performance eg: critical section [EE10], false sharing etc. From hardware side, multiple threads trying to use the shared architectural resource might result in contention for the resource [GKS12] eg: available memory bandwidth, shared caches, inefficient cache coherence policies etc. Therefore, parallel application performance depends both on the underlying architecture and the programming model used to port the application.

Many-core is expected to be a platform where one can exploit Instruction Level Parallelism, Thread Level Parallelism and Data Level Parallelism to gain maximum performance for a given application under a constant power budget. While determining the many-core performance, it is important to study these applications entirely because many effects are difficult to identify in a study that only evaluates kernels [HP03]. Amdahl's law [Amd67] indicates that the achievable performance of a parallelized application on a massively parallel many-core will significantly depend on the performance of the sequential code sections of the application. Currently emerging workloads contain mix of applications which ranges from being embarrassingly parallel to partially parallel (by partially parallel we mean that whole legacy code is not completely parallelizable) or completely sequential. These mix of applications place varied performance/power demands on the architecture. Simply replicating identical cores and the memory system into a many-core architecture (homogeneous many-cores) e.g Intel Xeon-Phi [JR13] does not solve the demands of these mixed workloads and hence, Heterogeneous Many-Cores (HMC) [[KFJ$^+$03], [SMQP09]] with complex cores targeting the sequential section and simple cores targeting the parallel section seems more adapted. Industry has already started attempting heterogeneous multi-core for saving power and to use the silicon die area efficiently, for example, ARM Big.LITTLE [Jef12] and in the near future heterogeneous many-cores will be available in the market.

In the many-core era, with lot of threads executing simultaneously it is critical to understand the performance bottlenecks that limits application scalability. Performance evaluation has been an interesting and challenging area of research in the multi-core architectures. It is highly used in making early design decisions for a yet to be built system. In the past few years, most of the performance studies focused on performance degradation occurring only in parallel section due to synchronization and communication between the threads [YMG14], [EE10]. And, number of interesting solutions have been proposed by migrating threads to take advantage of the cache locality and avoid data communication cost [LDT$^+$12], [SKM$^+$06], but, they did not focus on the impact on performance due to the serial section of the code in the multi-threaded program. This thesis studies the unexplored hardware and software bottleneck occurring due to explicit serial sections that were ignored in the multi-core era, but, will become very prominent and can be a bigger scaling limitation in many-core era.

Research questions

In this thesis, the research work focuses on answering the following research questions:

1. **Are the serial and parallel sections of the multi-threaded program same or different?** It is important to know whether the serial and parallel parts have a different micro-architecture independent characteristics like dynamic instruction mix, memory footprint, control flow (branches), memory access pattern etc. These information about an application or a group of applications will be the key to design the many-core with better area-performance trade-off.

2. **How to model application scalability for future many-cores considering the entire application?** Serial and parallel sections may not scale in the same order which the existing analytical models fails to capture. Therefore, a different performance evaluation technique that considers both serial and parallel scaling factor to determine the application scalability is required.

Contributions

The focus in this thesis is to show that the serial part of a multi-threaded program cannot be ignored and can be a performance bottleneck for some applications when executed on the many-core. Also, we study the difference between parallel and serial section of the program to design future many-cores with better area-performance tradeoff. The future many-core we consider in the context of this thesis can function independently with a host OS and not as an accelerator like currently available many-cores.

There are two main contributions in this work.

- As a first contribution, we analyzed a set of currently available multi-threaded applications and quantified the impact of serial section in the performance of the application in many-core era. To achieve this, we developed a methodology to monitor and measure the serial and parallel parts without instrumenting the program using a low overhead performance monitoring tool *tiptop* [Roh11]. Then, we proposed a more refined but still tractable, high level empirical performance model for multi-threaded applications, the Serial/Parallel Scaling (SPS) model to study the scalability and performance of applications in many-core era. SPS model learns the application behavior on a given architecture and provides realistic estimates of the achievable performance in future many-cores. Considering both input problem size and the number of cores in modeling, SPS model can help in making high level decisions on the design choice of future many-core architecture for a set of applications. We validate the model on two different architectures 1. Xeon Westmere cores with 24 logical cores and 2. Many-Integrated Cores (MIC) Xeon-Phi with 240 logical cores.

- As a second contribution, we characterized the inherent program behavior of the serial and parallel sections to find the difference between them. Then, using different simulation models, we explored the hardware requirements of the core that are needed for cost effective execution of serial and parallel sections. Our analysis shows that, the micro-architectural resource requirements of both these sections are different, thereby affirming that heterogeneous cores with few complex cores and many small cores will benefit most applications in many-core era.

Organization of the thesis

This thesis is organized as follows: Chapter 1 describes the evolution of hardware architecture from single-core to multi and many-core and discusses the current and future challenges in designing a many-core processor. Chapter 2 focuses on the different performance analysis techniques used and analyzes their limitations in many-core era. Chapter 3 introduces the SPS model and illustrates the inference obtained with the empirical model for different applications considered under study. Also, it focuses on showing the need of heterogeneous cores with our area-performance trade-off analysis. Chapter 4 studies the inherent program behavior of the serial and parallel sections and concludes that they might need different kind of processors in the many-core era. Finally Chapter 5 concludes this thesis by presenting a summary of contributions, and provides some direction for future work.

Chapter 1

Multi and Many-core processor architectures

In this chapter, we discuss the evolution of the microprocessor from single core to multi and many-core both from the technological and the architectural perspective. Then, we focus on the recent trends in the multi and many-core architectures designed in academia and industry. Finally, we discuss the current issues and future challenges in designing the many-core processors.

1.1 Rise and fall of single core era

From 1980's to 2002 the processor performance was growing exponentially due to the process technology scaling and micro-architectural innovations. This was the era of free lunch where you invest more on hardware and your software will run faster. This was mainly due to the exponentially increasing processor clock frequency on an yearly basis till 2004 [chr]. In this period, the industry had advanced from manufacturing processors at 3.0µm in 1977 to 90nm in 2004 [ITR15]. The belief behind the single core scaling was that: smaller the transistors, the more transistors in the same area, the faster they switch, the less energy they require, the less heat they dissipate and ultimately higher performance gains.

From the technology side, processor industry was driven by Moore's law [Moo98] which states that processor performance doubles every 18-24 months as the number of transistors in a dense integrated circuit doubles approximately every two years with new process technologies. From the architecture side, computer architects were able gain performance by improving the design choices [PJS97] and innovating new micro-architectural techniques. Design complexity increased initially with the introduction

[2][Edw] source for power, [chr] source for Clock frequency, [cou] source of Transistor count, [MIP] source for MIPS

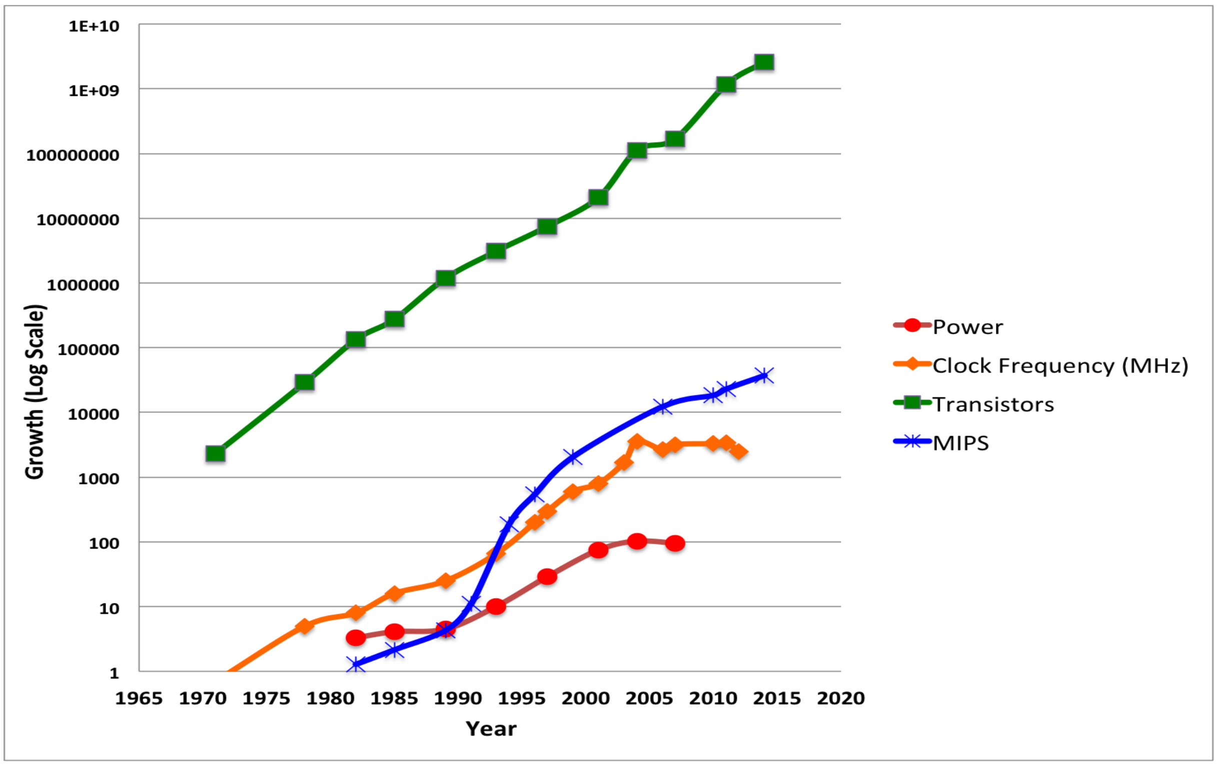


Figure 1.1: Graph showing different growth trends of INTEL microprocessor parameters from 1970 till 2014.[2]

of instruction pipelining where execution of multiple instructions can be partially overlapped followed by super-scalar execution where multiple execution units are used to execute multiple instructions simultaneously. Then, the recent developments like out-of-order execution using register renaming technique and speculative execution using branch prediction techniques became a standard component of single core high performance processors.

By the end of 2002, high performance processors were running at almost 4GHz [SHU+01]. Fast switching transistors along with architectural technique like implementing deeper pipelines [SC02] helped in achieving higher clock frequency which in turn resulted in higher processor performance. Figure 1.1 shows different trends as observed by Intel engineers which infers that 1980's to 2002 was the golden period of the single core processors. We can observe exponential growth in the amount of transistors in a single chip, clock speed of the processor, power consumed by the processor and the performance of the processor in this period.

Unfortunately, all good things have to come to an end and exponential growth of the processor performance hit a plateau by 2003. The reason behind this was mainly due to the end of Dennard scaling [DGR+74] i,e the supply voltage required to make the transistor work did not reduce and the power density increased [HAP+05] (amount of power required by transistors per unit area) that resulted in increased heat dissipation. This marked the end of single core era from the industry and the research towards improving single core performance slowed down in last decade. Following are the reasons that better advocate the performance saturation of the uni-processors:

- ***Power Wall*** - *Power consumption of transistors does not decrease as fast as density increases* [Bor99]. With higher clock frequency and more transistors in a given unit area, there will be more switches per cycle which results in higher power density and higher heat dissipation. In order to overcome power wall, power-efficient designs and power-management techniques were adopted which led to the saturation of the performance ([Pol99], [AML+10]).
- ***Memory Wall*** - *Memory speed does not increase as fast as computing speed*[WM95]. The rate of improvement of microprocessor speed exceeds the rate of improvement of DRAM memory speed. Therefore, various latency hiding techniques like data prefetching ([GHG+91], [CB92]), value prediction [ZC] are used to close the gap between the speed of computation and speed of data access.
- ***ILP Wall*** - *Law of diminishing returns on Instruction-Level Parallelism*[HP03]. Even with complex design, the achievable sequential throughput was low due to limitations in exploiting ILP. Limits of ILP has been studied for long now with first work from Wall *et al*[Wal91] considering the limitations imposed by register renaming, branch prediction and window size. They concluded that available ILP lies in the range of 5-7 instructions on standard benchmark programs. In recent research, different techniques of improving ILP are revisited using speculative execution and value prediction [PS13], [PS15].

Thus, *Power Wall + Memory Wall + ILP Wall = Brick Wall* [ABC+06] sums up the end of the single-core or uni-processor era. After 2002, most companies like AMD, IBM, INTEL, SUN stopped relying on the fact of improving the performance of uni-processors and started focusing on multi-cores ([BDK+], [Dal06]). A very familiar case was that Intel canceled the design of Tejas Pentium 4 processor in 2004[Wol04] as power consumption became a bigger issue. Nevertheless, there remained a way to make use of the still increasing number of transistors as dictated by Moore's law and that was to put more than one processor core on each chip. This marked the beginning of the multi-core revolution - *From Single Core to Multi-Core: Preparing for a new exponential* [PDG06].

1.2 Evolution of multi-core and many-core

There is no clear definition that defines the multi-core and many-core processors in both industry and academia. In the context of this thesis, we consider the following.

- **Multi-core** - It is just incremental to the single core by adding two or more cores (up to 10) on a single chip sharing one memory space under control of single OS. Mainly used for latency oriented or peak performance bound workloads.
- **Many-core** - The number of cores on a single die may vary from 10s to 100s depending on the size of the core. They may have uniform or Non-uniform memory architecture with a centralized or distributed OS. Eventually, $many - core$ is a buzzword coined after new Moore's law that states the number of cores per chip will double every two years over the next decade, leading us into the world of many-core processors. Many-cores are mainly used for throughput oriented workloads.

Free lunch is over

"Free lunch is over", "End of endless scalability" were very common statements in computer architecture industry in the early 2000's which meant that the exponential performance benefits reaped by the software developers just by upgrading their hardware was over - *'Do not rewrite software, Buy a new machine'*[Sut05]. This was possible because of the process technology evolution and micro-architectural improvements which resulted in higher computational power with more transistors on the same silicon die. After free lunch was over, in the past decade there was a major shift in the industry, where, both the software developers and hardware designers had to work together to see any marginal improvement in the performance.

Parallel architectures and parallel programming models became the main focus from 2005. From there on, over the past decade the performance gain were mainly due to software/hardware co-design by exploiting different sources of parallelism such as

- ***Instruction Level Parallelism (ILP):*** ILP was the main contributor to the performance in single core era by exploiting parallelism between independent instruction in a sequential program. In the age of multi and many-cores, it is still important to improve the single core performance focusing on ILP [SMYH12].
- ***Data Level Parallelism (DLP):*** DLP refers to the execution of the same operation on multiple datum simultaneously. In recent processors there are vector units which performs a single operation between elements of a vector to exploit DLP. Vector instructions are highly parallel and can be easily pipelined. For example MMX [PW96], SSE [Lom11], AVX [FBJ+08] instructions from INTEL,
- ***Thread Level Parallelism (TLP):*** TLP is the main contributor to the performance in multi and many-cores by running multiple flows of execution of a single process simultaneously in multiple cores on a different set of data. Simultaneous Multi-Threading (SMT) [TEL95] is the commonly used technique to exploit TLP and is used in almost all recent high performance processors.

These different source of parallelism can be realized either statically using compiler/software techniques or dynamically during the run-time using some hardware

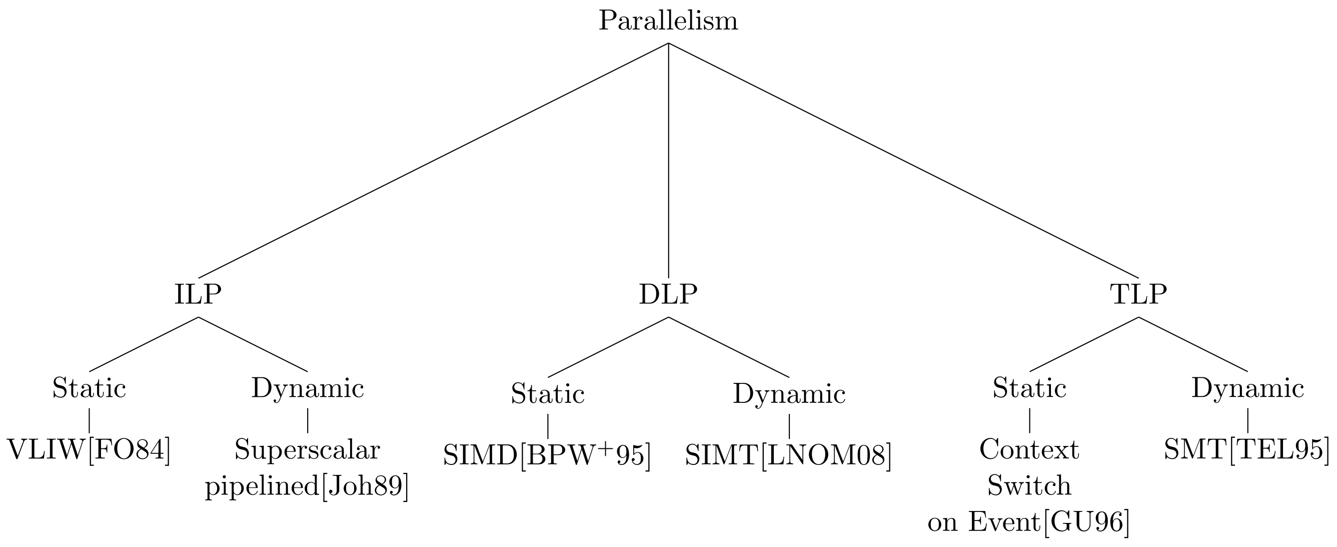


Diagram 1.1: Taxonomy of parallel architectures

techniques. Figure.1.1 clearly shows taxonomy of parallel architectures that exploit different kinds of parallelism. Current multi and many-cores evolved by integrating these different sources of parallelism in the architecture. In the next section, we discuss the current trend in the many-core architectures.

1.3 Trends in many-core architecture

In the last decade, different parallel architectures were implemented in both academia and industry which in someway or other tried to capture the potential performance gain using the above mentioned parallel taxonomy. The independent benefits of the 3 sources of parallelism are explicit but together how they perform in a single architecture is under study and highly depends on the application in hand. Combining the potential of all these source of parallelism, currently, there are 2 kinds of many-cores in the market.

1. **Few larger cores:** Ten's of large many-cores are used in server based applications which mainly targets on reducing the execution time of a workload at the expense of the power.

2. **Many small cores:** Hundred's or thousand's of many-cores are mainly used in throughput based application where amount of tasks processed per unit time matters the most.

The setup of these systems are very common: cores have a private small L1 instruction and data cache and some with a private L2 cache. In some architectures, cores are grouped in clusters of two or four cores, which connect via a low-latency interconnect to a shared L3 cache. Furthermore, the individual cores or clusters are connected via a NoC with a ring or mesh topology or a multilayer bus to each other, and via one or more memory controllers to external DDR memory. Table. 1.1 shows the recent implementations of the many-cores from both academia and the industry. These architectures all are single chip packages that delivers a high performance by utilizing multiple cores and different level of parallelism. We can observe that the processors belonging to few large cores runs at very high frequency. They are usually Out-of-Order complex cores with 2,4 or 8 way SMT features. On the other hand, we can see that the processors belonging to many small cores, runs at lower frequency around $1GHz$. They are usually simple In-Order cores with or without SMT.

We do not consider (general-purpose) GPUs at this point in our study. Although they have thousands of cores, they are predominantly used in streaming applications like graphics and specific scientific workloads and are mainly used as accelerator. Moreover, GPUs cannot work as a stand alone system with an operating system to manage the tasks running in it. As mentioned earlier, we consider the many-cores which can be programmed with existing conventional parallel libraries like MPI, OpenMP / Pthreads, can run an OS and has a conventional full fledged in-order or out-of-order core.

Processor	year	coretype	Freq. (GHz)	Cores (threads)
Few Large Cores				
IBM POWER 7 [KSSF10]	2010	C1	3.5	8 (32)
Oracle UltraSPARC T3 [STH+10]	2010	SPARC V9	1.6	16 (128)
Fujitsu SPARC64X [YMA+13]	2010	SPARC V9	3	16 (32)
Intel E7-8890V2 [int]	2014	Ivy Bridge-EX	2.8	15 (30)
Many Small Cores				
IBM CYCLOPS 64 [CZHG06]	2006	RISC	.5	80 (160)
Tilera TILE-Gx8072 [til]	2009	DSP	1	72
Intel SCC [HDH+10]	2010	pentium -100	1	48
Adapteva Epiphany-IV [Gwe11]	2011	RISC	.8	64
Intel Xeon-Phi MIC [JR13]	2012	Pentium -P54C	1.2	60 (240)
Kalray MPPA [Kal13]	2012	VLIW	.4	256
Intel Knights Landing [Ant13]	2015	Silvermont	NA	≥ 60

Table 1.1: History of many core in the academia and industry

1.4 Design challenges in many-core era

In this section, we review the known challenges which were faced with multi-cores that might probably continue to the many-core era and then discuss the forth coming challenges of the many-cores that are unexplored. In the past decade, as the number of cores in a single die grew, more scalability bottlenecks came into limelight from the parallel part due to the programming model, communication model, memory consistency model etc. As with the multi-core, the major challenges remain both with software and hardware in the many-core era.

1.4.1 Software challenges

Challenges faced during application development cycle of a parallel program are much more complex than their serial version. An usual parallel application development cycle consists of designing the parallel algorithm, implementing it in the form of a program, debugging, tuning and optimizing the code till best performance is achieved in the given hardware. Therefore, the first and foremost challenge moving to a multi and many-core is to identify the potential parallel algorithm to solve the problem or to identify potential areas in serial version of the program to parallelize. The software challenge arises during the implementation phase in choosing a right programming model, debugging and performance visualization tools.

1.4.1.1 Programming model

Programming models are important because they determine how well a range of different problems can be expressed for a variety of different architectures, and how efficiently they execute in a given architecture. In [VHvN+], the authors surveyed various available parallel programming models and concluded that, despite the large variety of programming models there are no model which a programmer can use in all stages from design to complete development of the parallel application. Moreover, they classified current day parallel programming models in 3 different classes:

1. Parallelism-centric models: These models are basically used to express the complex parallel algorithms. They are traditional models adopted for programming many-core processors. Eg: Pthreads [NBF98], MPI [GLS99], OpenMP [DM98] , Cilk [BJK+95] etc.

2. Hardware-centric models: These models are basically used to give a higher level abstraction to program the underlying hardware. These models require users to understand the chosen platform and parallelize applications for it. Eg: Intel TBB [Phe08], NVIDIA CUDA [K+07], AMD Brook+ [BFH+04], CellSS [BPBL06] etc.

3. Application-centric models: These models help users to find an effective parallel solutions for the given application and implement them with optimal granularity and parallel constructs to utilize the system efficiently. These models start with a clear design phase that transforms a generic application into a parallel one and are implemented using dedicated back end. Eg: SP@CE [VNGE+07], Sequoia [FHK+06] etc.

To be successful, programming models should be independent of the number of processors [ABC+06] and should be portable among different parallel architectures. The programmability gap is the difference between the theoretical performance of a platform and the observed performance of the applications running on that platform. The advent of many-core widens the programmability gap because platform peak performance is achieved only if applications are able to extract and express multiple layers of parallelism, at par with those offered by the hardware platform.

1.4.1.2 Debugging and Tuning

Testing, debugging, maintenance and performance tuning of software applications usually take up more than 50% of the total effort that goes into designing software systems. In the context of many-core systems, it's critical to record per-thread and per-core behavior as well as the interaction between different shared resources and impact on the usage of shared hardware components. The obvious challenge is dealing with the ever increasing number of threads and cores. There will be scaling issues for the performance-monitoring tools as they use sampling technique to monitor lot of different PMU events. Added to this, when they monitor PMUs on per-thread basis these tools should scale well to monitor concurrently in 240 logical cores/threads in many-core processors like Xeon-Phi. Though all the hardware vendors provide their own performance analysis tools like Intel VTune [Mar11] , AMD CodeAnalyst [DTC08], NVIDIA CUPTI [MBS+11] etc, the challenges lie ahead in making them a low overhead tool in the many-core era. In our experience, VTune was crashing when used for more than 192 threads in Intel Xeon-phi and also the Tiptop [Roh11] tool which we have used in this thesis initially did not scale well. We had to modify the code to reduce the overhead and validate it.

1.4.2 Hardware challenges

With the number of cores in a single chip increasing, there are predictions that, in future, not all the cores can be powered on simultaneously in the chip due to the thermal and power constraints. Those parts of the die which cannot be powered on are called "Dark Silicon" [EBSA+11]. Basic hardware challenge comes from the dark silicon problem which describes the limitations and challenges of many-core from the technology perspective. Other than that, there are other design/architectural issues and challenges which hinder the application scalability in the current multi- and many-cores. The issues are mainly with the capability of the cores, communication among the cores and communication with the memory system.

1.4.2.1 Core model

In the taxonomy of parallel architectures (Figure 1.1), we saw that there are three different types of exploitable parallelism. One of the main challenges faced by many-core architects is how to design a chip with best performance/power/area trade-off by employing all the three sources of parallelism. Most current general-purpose multi-core processors are homogeneous both in terms of Instruction Set Architecture (ISA) [KFJ+03] and the core architecture [SMQP09] used. All the different architectures presented in Table 1.1 are made up of homogeneous cores which are simple replication of a base single core with an on-chip interconnect to create a many-core processor. The emerging mix of workloads requires lot more dynamic hardware than the homogeneous many-cores to attain high performance under constant power and thermal budget and hence the need of heterogeneous cores. ARM introduced Big.LITTLE which was the first heterogeneous multi-core processor in a single die that mainly targeted the low-power multi-core segment. But, many-core heterogeneity is still not explored completely and hence there are enough opportunities to explore and find an optimal design which can achieve better performance/area/power goals.

1.4.2.2 Core to Core Communication

Having multiple cores on a chip with private and shared caches requires inter-core communication mechanisms to maintain coherence on the shared data in the whole memory system. Core to core communication for a long time in the shared memory multiprocessors happened through a common bus with on-chip hardware or software cache coherence protocols. Although coherence protocols serves the purpose in today's multi-core systems, the conventional wisdom is that on-chip cache coherence will not scale to the large number of cores in the many-core era due to increasing traffic and latency. To combat the issue due to the interconnect, research community is exploring the possible options of using easily scalable, high-performance and low power Network on Chips [BKA10]. Also, determining the communication pattern from the application behavior [BWFM09] will help in building better communication systems for future many-cores. In the currently available many-core platforms, crossbars are used in GPUs which has limited scalability and uniform latency, ring networks are used in the recent Intel Xeon processors.

1.4.2.3 Memory bandwidth

Memory bandwidth will be another scalability bottleneck in the many-core era. A many-core system with thousands of cores will demand 100's of GBs of memory bandwidth, and a traditional memory subsystem solution is not sufficient to handle high data traffic. More over, the memory link becomes a shared resource and memory-controllers are required to arbitrate the request and to avoid contention. Location of the memory controllers within the on-chip fabric also plays an important role in the performance of memory-intensive applications [AEJK$^+$09].

1.4.3 Impact of serial section

All the above mentioned challenges mainly focused on the design challenges of the parallel kernels which is only part of the application. When we consider the entire application, there are more pressing issues to be addressed. Currently available many-cores are more of accelerators which enhances the performance of applications having high parallelism. But even then, there are certain parts of the application which cannot be parallelized or left unparallelized due to ignorance. In such cases, the serial part in the program cannot be neglected. Amdahl's law [Amd67] determines the theoretical speedup of an application by considering a fixed amount of work done across varying number of cores. The main observation from the model is that, speedup mainly depends on the size of the serial portion even if we assume infinite parallel computing resources i.e, the execution time of a parallel application for a given problem size cannot be reduced below the execution time of the serial part of the program. Amdahl's speedup is given by Eq. 1.1, where f stands for the fraction of parallel part in the program, and P is the number of cores of the machine on which the application is executed. *In simple terms, for a given application, the maximum achievable speedup is determined by the fraction of serial part of the program that remains constant for a workload. Moreover, Amdahl's law does not define what contributes to the serial part of the program.*

$$speedup_{Amdahl} = \frac{1}{(1-f)+\frac{f}{P}} \tag{1.1}$$

The solution to this problem is to speedup the serial part as much as possible to improve the performance of the whole application. We believe that in future, many-core applications will have bit of legacy code which will maintain the control flow of the parallel kernels and this part of the program will not be negligible along with some computation which happens in single thread. Therefore, the future many core will be heterogeneous with few large cores and many small cores in a single die.

1.5 Future Heterogeneous Many-Cores (HMC)

Heterogeneity can be in different forms in many-core like core size, memory system, ISA, OS etc. HMC's are made for different goals which can be classified into two 1. Performance/power optimized HMCs, 2. Latency/throughput optimized HMCs. In this thesis, we mainly focus on the heterogeneity with respect to the core size and the research falls under the first category. Core size directly relates to the performance capabilities of the core, therefore the HMC designed should have optimal area-performance trade-off. The large cores are usually complex super scalar processors which targets on exploiting ILP. These cores feature high-end micro-architecture design choices such as deep pipeline, larger issue width, out of order execution, powerful prefetchers and branch predictors. On the other hand, set of small cores are simple processors which targets on exploiting TLP if the application can spawn multiple threads that can work simultaneously. These cores feature shallow pipelines, less aggressive branch predictors and lesser number of execution units. These small cores are highly power efficient. Therefore, the sequential section of a multi-threaded program will benefit if executed in larger cores and the many parallel threads will benefit if executed in the smaller cores. One of the main challenges of future HMCs will be to cover the performance improvement on the entire application as compared against today's homogeneous cores / hardware accelerators which just focuses on improving the parallel and computation intensive kernels. In this thesis, we address this issue by considering the entire application in our performance evaluation.

1.6 Summary

For almost three decades starting from 1970s processors were primarily single core. CPU manufacturers achieved higher performance by increasing operating frequencies, core sizes, and also by using advanced manufacturing processes allowing more transistors in the same chip area. But by the early 2000s, they realized that the continuous increase in frequency and core sizes caused exponential increase in power consumption and excessive heat dissipation. Thus, the industry started manufacturing multi-core CPUs to deliver higher performance with a constant power budget which reflects in today's desktops, laptops and even smart phones. Both industry and research communities have already been working on the many-core for sometime now after the single core era ended.

In this chapter, we discussed the current trend in multi- and many-core architectures. We then classified currently available many-cores into 1. few large many-cores that are used to achieve best performance on latency oriented applications and 2. many small many-cores that are used to achieve best performance on throughput oriented applications. Then, we discussed the hardware and software challenges faced by computer architects and programmers in exploiting many-cores. Most of the challenges we discussed were on identifying and resolving the bottlenecks appearing in the parallel kernels of the multi-threaded application and neglected looking at the other possible bottlenecks that may arise when entire application is executed in the many-core system. With our belief that the future parallelized codes will still have some sequential parts, we then introduced a bit about the impact of serial scaling in many-core era which is one of the research goals in this thesis. Also, we finally discussed utilizing the heterogeneous many-cores as a possible solution to overcome the performance limits posed by the scaling serial section of the multi-threaded applications.

In the next chapter, we will focus on different performance evaluation techniques employed in determining the parallel applications performance and will introduce the state of the art of parallel analytical models that are used to determine the performance of applications executed on parallel architectures.

Chapter 2

Performance evaluation of Many-core processors

In this chapter, we focus on the performance analysis of the parallel applications in the multi- and many-core era. First part of this chapter focuses on different kinds of performance analysis techniques employed till now in the computer architecture community and discuss their strengths and weakness. Then, we present the state of the art existing parallel models that are used to estimate the performance of parallel programs and discuss their limitations in many-core era.

Performance evaluation of a system is a very important step in building any complex system. Performance evaluation techniques in the field of computer engineering can be used in any phase of the systems life-cycle, right from design phase till maintenance phase of the finished system. During the design phase, it can be used in exploring the design space to choose the right design parameters. During implementation phase, it can be used to predict the performance of a large system by experimenting in the smaller prototypes. In the maintenance or upgrade phase of an existing system it can be used to quantify the impact of possible upgrade on the system. Therefore, performance evaluation holds a definite role in the field of computer architecture and system design. Performance evaluation techniques can be classified into two:

1. **Performance modeling:** Modeling is usually carried out before the prototype is built. By modeling the system, several design decisions can be drawn which can avoid unexpected problems while prototyping. Performance models can be further classified into 1) simulation based models and 2) analytical models. Table 2.1 gives a brief outline of different performance modeling techniques used in computer science.

2. **Performance measurement:** Measurement happens after the prototype is built. By measuring the performance in the prototype we can validate the models used in the design process and provide feedback for the improvement of future

designs.

2.1 Performance evaluation techniques

Several techinques are available that can be used in evaluating performance of a system. And, selecting a performance evaluation technique depends on different parameters like Stage (phase of the life-cycle of the system), Accuracy (reliability of the results), Cost (resource required) and Time (time required for the evaluation) as shown in Table.2.1. We further explore these different techniques in detail in this section and discuss on the specific techniques utilized in the context of this thesis.

2.1.1 Modeling

Performance modeling becomes a prime tool in the early stages of the design process, when the actual system is not yet built. It can be used by the performance analyst to determine the maximum achievable performance of a system and also to verify the correctness of the design choices while building the system. Modeling can be done either by simulation or analytically.

2.1.1.1 Simulation

In micro-processor industry, developing a prototype is a very tedious, expensive and time consuming job. Therefore, most of the processor design cycle starts with a simulation model. They are usually written in high level languages like C/C++ or java. Computer architects use simulators mainly to model accurate future systems or to make learned decision on the type of components to be used in the processor or to study the impact of some parameter in the future processors. Software engineers use simulators for the early development of the software even before the prototype is ready.

A computer architecture simulator is usually split into two. 1. Functional model and 2. Timing model. Functional model models the Instruction Set Architecture and the different hardware components present in the processor. They are used to simulate the functionality of the target processor. Timing model models the performance of the processor by evaluating the timing details at a cycle accurate level. Generally, these simulators with both functional and timing models are modeled with lot of details for accuracy and tend to be slow.

In contrary, we can also build low overhead pure functional model simulators which can run faster than usual cycle accurate simulators. It can be used for design space exploration studies and behavioral studies of the hardware components. Pure functional simulators implements only the functional model to validate the correctness of an architecture and to obtain some abstract performance metrics which do not depend on timing information. For example: to study branch prediction accuracy, cache hit ratio etc of a program under different hardware constraints does not require timing

Evaluation Technique	Types	Stage	Time required	Accuracy	Cost
Simulation	**Trace Driven Simulation** Execution Driven Simulation Complete System Simulation Event Driven Simulation Statistical Simulation	Any	More	Moderate	Medium
Analytical Model	Mechanistic Model **Empirical Model** Probabilistic Model	Any	Less	Low	Low
Measurement	**On chip Performance Monitoring Units** Off chip hardware monitoring units Code instrumentation	Prototype	Varies	Varies	High

Table 2.1: Classification of performance evaluation techniques along with different trade-offs each technique has which will be used as a selection criterion. *Stage* refers to the stage of the design cycle the technique is used, *Time required* is the analysis time required, *Accuracy* is how much the results are reliable, *Cost* refers to the resource required for using that technique.

model. There are various simulation techniques which can be adopted depending on the nature of study as shown in Table.2.1 that are further explained below

1. *Trace driven simulation*: This method uses instruction trace of the program executed on a real system as simulator input. Therefore, all required fields of information needs to be collected with respect to the goal of the study. For example, a cache simulator that simulates memory hierarchy like Cachesim5 [CK94] and Dinero IV [EH] requires information like addresses of the instruction and data references. Advantage of this method is that the same trace can be utilized by the research community and the results can be replicated. Disadvantages are: 1. if the program executes longer, then, large amount of trace data has to be stored. This can be overcome to a certain extent by using data compressing techniques. 2.Trace files stores only the information of the committed instructions and hence will loose the information from the mispredicted path in the case of speculative execution and hence cannot be simulated.

2. *Execution driven simulation*: These simulators use the program executable as simulator inputs. In this method not all the instructions are executed by the simulator, i,e only the instructions that are of interest are simulated. This is done by inserting calls to the simulator routine in the application program. Parts of the program marked by simulator calls are executed in the simulator and other instructions are executed in the host system. Advantages are: 1. Size of the input is small as they are only the static instructions present in the program executable and 2. mispredicted path can be simulated as well. In contrary, long simulator development cycle and reduced platform portability are disadvantages of this simulation technique. SimpleScalar [ALE02] is a very widely used execution driven simulator.

3. *Complete system simulation* Many trace and execution driven simulators simulate only the application and does not account the interference of the input-output (I/O) and Operating System (OS) activities in the results. Accounting I/O and OS activities becomes important for performance analysis of multi-threaded programs where OS libraries are used. Advantages are: 1. full system performance can be obtained and 2. cycle accurate. Disadvantage is that they are very difficult and time consuming to develop. Some of the popular complete system simulators are SimOS [RHWG95] and SIMICS [MCE+02].

4. *Event driven simulation*: Above described three simulators are cycle-by-cycle simulators i,e all the stages of the processor are executed and the processor states are updated every cycle. In an event-driven simulation, tasks are posted to an event queue at the end of each simulation cycle. During each simulation cycle, a scheduler scans the events in the queue and services them in the time-order in which they are scheduled for. For example, if the current simulation time is 1000 cycles and the earliest event in the queue is to occur at 1100 cycles, the simulation time is advanced by 100 cycles for the event to occur at 1100 cycle

instead of simulating all the cycles in between. This simulation technique is not very common in computer architecture simulators but are widely used in VHDL based simulators which simulates the hardware at logic level.

5. *Statistical simulation*: In this simulation technique, a statistically generated trace is used as a simulator input. Initially a benchmark program is analyzed to find their inherent program characteristics and synthetic programs with same characteristics are generated [EBJS+04]. The processor is modeled at a reduced level of details and can be faster than other simulation techniques. For example, cache accesses may be considered as hits or misses based on a statistical profile without actually simulating a cache.

Later in this thesis (in Chapter 4), we have used simulation techniques to characterize and explore the processor core design space of serial and parallel section of the multi-threaded programs. We have adopted trace based simulation technique as it is easier to build a simulator depending on the exploration requirements. For example, to study the kind of branch predictors required in the core executing serial and parallel sections of a multi-threaded program, a detailed cycle accurate simulator is not required but a simple simulator which analyzes the control trace of the program is sufficient.

2.1.1.2 Analytical Modeling

Analytical modeling is a mathematical modeling technique that uses a set of equations to describe the behavior of a complex system. It can be of any form like dynamic system, statistical model, differential equations or game theory etc. There are very few works on analytical modeling of microprocessors as they are less accurate compared to the cycle-accurate simulators. On the other hand, it is sufficient for early design evaluations of multi and many-core systems as they are used to drive the future trend at the system level. Widely used analytical models in the field of computer science are mechanistic model, empirical model and probabilistic model.

Mechanistic models are built out of the basic understanding of the mechanics of the underlying system and provides fundamental insights about the system. It is also known as White-box-approach as the model explains the behavior of the complex system. There were few models for single core processor micro-architecture using mechanistic models. For instance, in [MSJ99], the authors built a mechanistic model of the instruction window and issue mechanism in out-of-order processors to gain insight in the impact of instruction fetch bandwidth on overall performance. Further improving that, there are some mechanistic first-order model that provides insights into the working of a super-scalar Out-of-Order processors ([KS04], [EEKS09]) and In-order processor models ([BEE15] [BEE12]).

In contrary, empirical model does not require much knowledge about the underlying system. Main idea behind empirical modeling is to learn or infer the performance model using statistical methods and machine learning techniques like regression, neural networks etc. Empirical model is easy to build but lacks the insight details compared to the mechanistic models and needs lot of experimental data to analyze and validate the system. Hence, it is also known as black-box approach. When it comes to the system level study, this model is preferred more than mechanistic models. Some prior work that studied the single core processor performance using empirical models are ([DJO07] [IMC$^+$06] [JVT06a] [JVT06b] [LB06]). In this thesis, we have used the empirical model to predict the execution time of an application in the many-core era by learning the application behavior in an existing architecture and extrapolating how the application will scale if the architecture is scaled proportionally. In simple terms, our empirical model presented in Chapter 3 can be used to study the application scalability.

Currently available performance models to study high level performance of multi- and many-core processors are probabilistic models ([Amd67], [EE10], [Gus88], [HM08], [YMG14]) which are very general. These models are further explained in detail in Section 2.3.

2.1.2 Performance measurement techniques

Performance measurement techniques are used to validate the system once the prototype is built. Also, to tune and optimize the application ported to the system and to influence the future design of the system by understanding the bottleneck. Usual measurement techniques involves utilization of the on-chip Performance Monitoring Units (PMUs) [Spr02] available in the processor to study different performance metrics like IPC, cache miss ratio, branch misprediction ratio etc. Architects use PMUs to extract application characteristics and propose new hardware mechanisms; compiler developers use PMUs to study how generated code behaves on particular hardware; software developers use PMUs to identify hot spots in their applications and to tune the application for the best performance in the platform.

The number of counters available (i,e hardware registers used for collecting performance data) varies among different processors. Multiple events can be monitored by either time based sampling where the tool will interrupt the processor at a regular sampling interval to collect event counts and to reset the PMU counters if necessary or event based sampling where the counters are sampled based on the specified event values. *Perf* [dM10] is a profiler tool for Linux 2.6+ based systems which is present by default in the recent Linux kernels as *perf_events* library. It provides a simple command line interface to collect and analyze the application performance by polling the PMUs. LIKWID [THW10] is another set of command-line tools which supports monitoring of multi-threaded and hybrid shared/distributed-memory parallel code running in a multi-core system. LIKWID addresses four key problems: Probing the thread and cache topology of a shared-memory node, enforcing thread-core affinity on a program, measuring performance counter metrics, and toggling hardware prefetchers. There are other tools which provides APIs to access and collect information from the PMUs but these tools requires source code modification. PAPI [MBDH99] is a library that provides operating system independent access to the hardware and provides an API to programmers for setting up and reading the counters. LiMiT [DS11] is a very recent proposal to reduce the overhead of reading counters, by directly reading machine registers and avoiding the system call. PAPI and LiMiT both require access to the source code of the application to be monitored. LiMiT is faster with low overhead, but requires changes to the Linux kernel with a patch. Other open-source tools include OProfile [Lev04] and perfmon2 [Era03] . Also, processor vendors provide proprietary performance monitoring tools such as Intel offers the VTune [Rei05] performance analyzer, IBM developed WAIT [AAFM10]- a tool to diagnose idle time in commercial workloads etc.

In this thesis, we used Tiptop [Roh11] to obtain the run-time measurements from the Performance Monitoring Unit (PMU) and LIKWID [THW10] to pin the threads to a particular core (Chapter 3). Tiptop is a command-line tool for the Linux environment which is very similar to *top* shell command. Tiptop is built with the perf_event system library which is available in the Linux kernel. This system call lets Tiptop register new counters for process running on the machine, and subsequently reads the value of the counters. Tiptop monitors all the necessary parameters of an attached process and periodically logs the values from the Performance Monitoring Unit which are then processed to get the desired metrics. Tiptop works on unmodified benchmarks and has only very marginal performance impact thus avoiding modification to the code. Events can be counted per thread and can be sampled at coarse-grain granularity to study the behavior of the multi-threaded applications.

2.1.3 Program profilers

"Program analysis tools or profilers are extremely important for understanding program behavior. Computer architects need such tools to evaluate how well programs will perform on new architectures. Software writers need tools to analyze their programs and identify critical sections of code. Compiler writers often use such tools to find out how well their instruction scheduling or branch prediction algorithm is performing or to provide input for profile driven optimizations" [SE04]. The utility of such a profiling tool was demonstrated by INTEL using PINTOOL [LCM$^+$05] to profile and analyze the micro-architecture independent program [HE07] behavior like instruction mix, register reuse distribution, memory access distribution etc of multiple applications which were used to classify and cluster applications. Such profilers often accept program executable as input and analyze each retired instruction of the application. Program profilers usually add some instrumentation code which will enable the run-time data collection. This instrumentation can happen either at source code level or at compiler level (AE [Lar90], Spike [CG95]) or even at binary or executable level (SPIX [CG95] and SHADE for SPARC [CK94], nixie for MIPS [PSM95], ATOM [SE04] for Alpha). Much advanced profilers performs instrumentation even during run-time (PINTOOL [LCM$^+$05] for IA32). Moreover, these tools can have an overhead while profiling the program which can cause execution time dilation. Hence, enough care should be taken to ensure that the tool has a low overhead and utilizes minimum resource of the processor. These program profilers can also be used to take trace of the analyzed instructions and feed it as input to the trace based simulator or as the front end of simulators. In this thesis, we have used PINTOOL as a front end for our trace based simulators to explore the design space of the parallel and serial sections of a multi-threaded program in Chapter 4.

2.2 Benchmarks

Performance measurements helps us understand the bottleneck in the system. In order to perform meaningful measurements, the workload should be carefully selected. Benchmarks to be used for performance evaluation have always been controversial. It is extremely difficult to define and identify representative benchmarks. In [PJJ07], Lizy *et al* analyze consequence of the the partial usage of benchmark suites by researchers (due to simulation time constraints, compiler difficulties, or library or system call issues) and concluded that random subset can lead to misleading results. Computer system benchmarks are from different categories such as CPU benchmarks like SPEC [Uni89], NAS [JFY99], PARSEC [BKSL08], SPLASH [WOT$^+$95] etc. Embedded and media benchmark such as EEMBC [Poo07], MediaBench [LPMS97], MiBench [GRE$^+$01] etc. Java based benchmarks like SPECjbb2000, SciMark [BSW$^+$00], MorphMark [mor] etc. Transaction Processing Benchmarks [TPC] such as TPC-C, TPC-H, TPC-R, TCP-W etc. In this thesis, our goal is to understand the difference between serial and parallel section of a multi-threaded program and hence we chose parallel benchmark suite like NAS [JFY99], PARSEC [BKSL08], SPLASH [WOT$^+$95], RODINIA [CBM$^+$09] and LONESTAR [KBCP09] for our study. The benchmarks used in particular are further discussed in Chapter 3 and 4 respectively.

2.3 Existing Parallel program performance models

In this section, we discuss the basic laws of parallelism that have influenced the parallel computing research in the past few decades i,e Amdahl's Law and Gustafson's Law along with different variations of them in the recent research which are applicable in the many-core era. We first discuss these 2 laws in detail before their multi-core versions.

2.3.1 Amdahl's Law

Amdahl's law [Amd67] determines the theoretical speedup of an application by considering a fixed amount of work done across varying number of cores, as shown in Fig. 2.1. Amdahl's speedup is given by Eq. 2.1, where f stands for the fraction of parallel part in the program, and P is the number of cores of the machine on which the application is executed. If P tends towards infinity, then the term f/P tends towards zero and only the serial component $1-f$ of the program will be left. The main observation from the model is that, speedup mainly depends on the size of the serial portion even if we assume infinite parallel computing resources, i.e., the execution time of a parallel application for a given problem size cannot be reduced below the execution time of the serial part of the program. *In simple terms, for a given application, the maximum achievable speedup is determined by the fraction of serial part of the program that remains constant for a workload. Moreover, Amdahl's law does not define what contributes to the serial part of the program.*

$$speedup_{Amdahl} = \frac{1}{(1-f)+\frac{f}{P}} \tag{2.1}$$

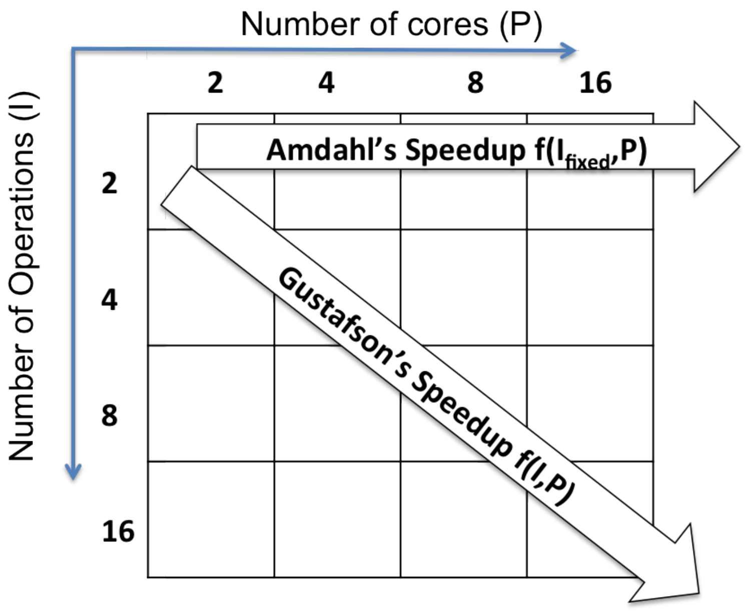


Figure 2.1: Amdahl's law assumes a fixed workload while Gustafson's law assumes scaled workload.

Through fixing the fraction of sequential code in an application, and considering that it cannot vary, Amdahl's law implies that there are no possibilities to increase the performance of an application above a certain limit through parallelization. Achievable Amdahl's speedup is shown in Figure 2.2. We can infer clearly that even for an application which is 99% parallel the maximum achieved speedup saturates around 90 when executed on 1024 cores and for a 90% parallel program the speedup saturates at 128 cores with a achievable speedup of 10. Amdahl's law is very pessimistic[1] and this is clearly not always the case: more cores may enable speculative and run-ahead execution of the sequential part, resulting in a speedup without actually turning the

[1]Increasing the degree of parallelism reduces the data set of each core, making it more likely to fit in the private cache. So sometimes the speedup becomes superlinear which is not captured by Amdahl's law.

sequential code into a parallel one. Amdahl's law considers the processing unit as the only component in the model but in today's processors the core is made up of processing units and a private cache. The best performance is obtained if the input data set of the program perfectly fits in the cache i,e performance of a program depends on the availability of data in the cache for the immediate computation. Therefore the solutions are either to use a bigger cache or an alternative solution of using prefetchers to bring the required data into the cache at right time. This particular case was also researched as a part of DAL project where small cores in a many-core can be used to execute helper threads - generate prefetch for memory intensive sequential threads on large core and improve the performance of serial program [SKS14].

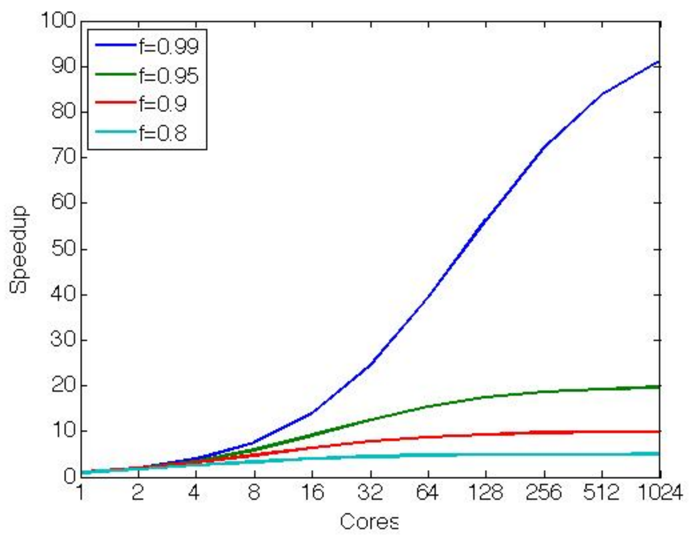


Figure 2.2: Amdahl's law

2.3.2 Gustafson's Law

Gustafson's law [Gus88] determines the theoretical speedup of an application by considering a varying amount of work done across varying number of cores, as shown in Fig. 2.1. Gustafson's law assumes implicitly a very different scheme for parallel execution. *Gustafson's law assumed that the parallel part of the application increases linearly with the workload size while the serial part remains constant and eventually becomes negligible with large workloads.* In simple terms, when the problem size scales the parallel part scales faster than the serial part and hence serial part becomes negligible. According to him, we should use larger system to solve larger problems in constant time i,e we can achieve linear speedup if we scale the workload size and the number of processing units proportionately. Gustafson's speedup is given by Eq. 2.2, where, f is the fraction of time spent on executing the parallel part of the application on P processors and the rest $1 - f$ is spent on the serial part. Achievable Gustafson's speedup is shown in Figure 2.3. We can infer that this law is very optimistic as the application which is 99% parallel can achieve a near linear speedup of 1000 when 1024 cores are used and even an application which is 80% parallel can achieve a maximum speedup of 800 on a system with 1024 cores. Along with serial section, Gustafson's law also neglected the possible scaling bottleneck like communication and synchronization which makes the law optimistic.

$$speedup_{Gustafson} = (1 - f) + f * P \tag{2.2}$$

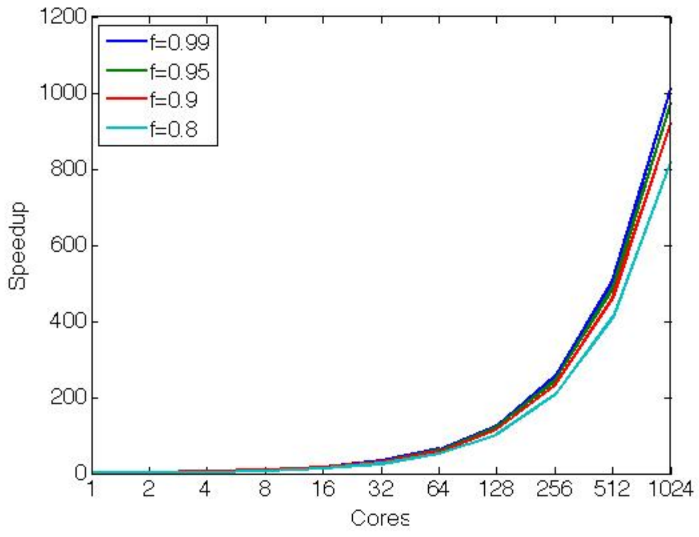


Figure 2.3: Gustafson's law

2.3.3 Amdahl's Law for multi-core era

Further, extending the Amdahl's passive model, Hill *et al* [HM08] proposed a performance-area model called *Amdahl's law in the multi-core era.* Their model used an abstract quantity called Base Core Equivalent (BCE) which is the cost of the baseline core. Their model had few assumptions like the base line core i,e 1 BCE includes only the core components like ALUs and L1 private cache. Rest shared components like lower level caches, interconnects and memory controllers does not belong to the BCE area. They also assume that, in the future, micro-architectural techniques to dynamically combine smaller cores into a larger core will be available and simple to implement. There are already couple of related works in the academia which affirms that the dynamic fusion of cores are possible and are highly energy efficient. Kim *et al* [KSG$^+$07] and Ipek *et al* [IKKM07] introduced techniques to compose several cores into a larger one. Khubaib *et al* [KSH$^+$12] introduced MorphCore, a hybrid processor that can run a single OoO thread or several in-order threads. Lukefahr *et al* [LPD$^+$12] proposed composite cores, where, two different back ends are integrated in the same processor and a controller switches between OoO or in-order back ends depending on the performance/power ratio. Carlos *et al* [CJRP14] proposed Yoga: A Hybrid Dynamic VLIW/OoO Processor which can switch between VLIW and OoO mode depending on the costs and benefits at each point in time. This implies that architects can utilize r BCEs to create a core with a sequential performance of $perf(r)$. Their final assumption was that the $perf(r)$ of the core made of r BCEs will be equal to $\sqrt{r}$ which was an observation of the prominent Intel technologist Shekhar Borkar [Bor07].

In the extended model, Hill *et al* analyzed the achievable theoretical speedup in 3 different configuration 1. Symmetric multi-core where all cores are similar, 2. Asymmetric multi-cores where one core is bigger and powerful, rest other cores are small and simple and 3. Dynamic multi-core where number of small and simple cores can dynamically combine to form a larger core with better performance. The given size of silicon die can contain at-most n BCEs and speedup is reported relative to execution speed on a core built with one BCE as shown in Equation 2.3.

$$Speedup = \frac{Executiontime_{1BCE}}{Executiontime_{nBCE}} \tag{2.3}$$

A symmetric multi-core chip with a resource budget of n = 16 BCEs, for example, can have sixteen cores of 1-BCE each or four cores of 4-BCE each, or, in general, n/r cores of r-BCEs each. Therefore, the symmetric speedup is given by the Equation 2.4 where the speedup depends on parallelizable fraction f, the total chip resources in BCEs (n), and the per core resource in BCEs (r). The chip uses one core to execute serial part at performance $perf(r)$ and rest all n/r cores to execute the parallel parts at performance $perf(r) \times n/r$. Symmetric speedup can be obtained using the Equation 2.4 and figure 2.4 shows how the speedup varies (along Y-axis) for different parallel fraction f over different BCE size (along X-axis). The interesting observations is that, the achievable theoretical speedup shifts towards using bigger core as the serial section in the program increases. For example, when $f = 0.99$, the peak speed up is achieved when each core was of size 8-BCEs, and as the parallel portion decreases to $f = 0.8$ i,e 20% of the program is serial, we can observe that the peak theoretical speed up is achieved with every core of size 256-BCEs.

$$Speedup_{symmetric}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f*r}{perf(r)*n}} \tag{2.4}$$

An asymmetric multi-core with a resource budget of n = 16 BCEs, for example, can have one 4-BCE core and twelve 1-BCE cores or one 9-BCE core and seven 1-BCE cores and so on. Thus, in an asymmetric multi-core, the single large core uses r resources and leaves rest n-r cores to be 1-BCE cores. Cumulatively, an asymmetric multi-core will have $1 + n - r$ cores. Asymmetric speedup can be obtained using the Equation 2.5 and from figure 2.5, we can observe that the maximum achievable speedup is $\approx 4x$ more than the symmetric case when $f = 0.99$. This is mainly because the one big core

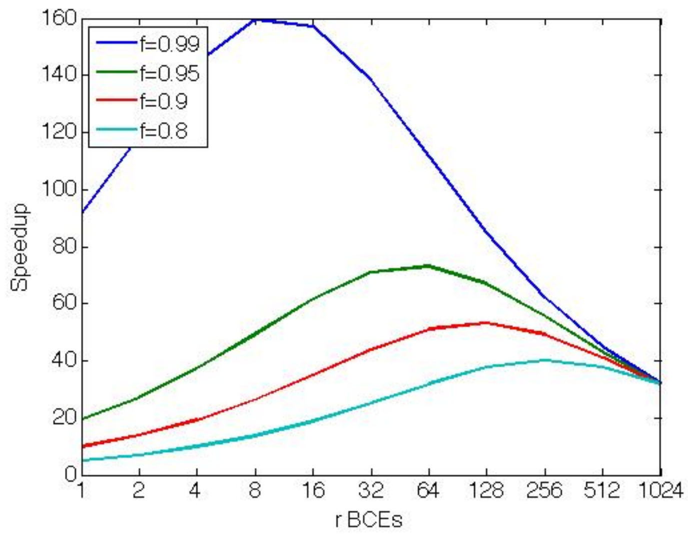


Figure 2.4: Amdahl's law in multi-core era with symmetric configuration by varying r from 1-BCE to n-BCE with n=1024.

executing the serial section is more powerful and the rest $n - r$ small cores provide sufficient core density to execute parallel parts.

$$Speedup_{asymmetric}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{perf(r)+n-r}} \tag{2.5}$$

A dynamic multi-core with a resource budget of n=16 BCEs, for example, can have 16 small 1-BCE when executing the parallel section and can dynamically combine to form a big n-BCE core with $r = n$ while executing serial section. In sequential mode, this dynamic multi-core chip can execute with performance perf(r) by combining smaller cores dynamically to form a larger core of r BCEs. In parallel mode, a dynamic multi-core gets performance n using all base cores (1-BCE) in parallel. Dynamic speedup can be obtained using the Equation 2.6. Dynamic cores can achieve higher speed up than the asymmetric core (shown in Figure 2.6) provided the core combining technology becomes feasible and the applications execute serial and parallel part in longer phases.

$$Speedup_{dynamic}(f, n, r) = \frac{1}{\frac{1-f}{perf(r)} + \frac{f}{n}} \tag{2.6}$$

Eyerman *et al* [EE10] further extended the Amdahl's law in many-core era with a probabilistic model which captures the effect of Critical Section(CS) in the parallel part as a contribution to the serial section of the program. It tries to address the case that, in addition to regular serial section, synchronization effects also degrades the performance of a parallel program. The sequential part contributed by the CS is determined by the probability for entering a critical section and the contention probability

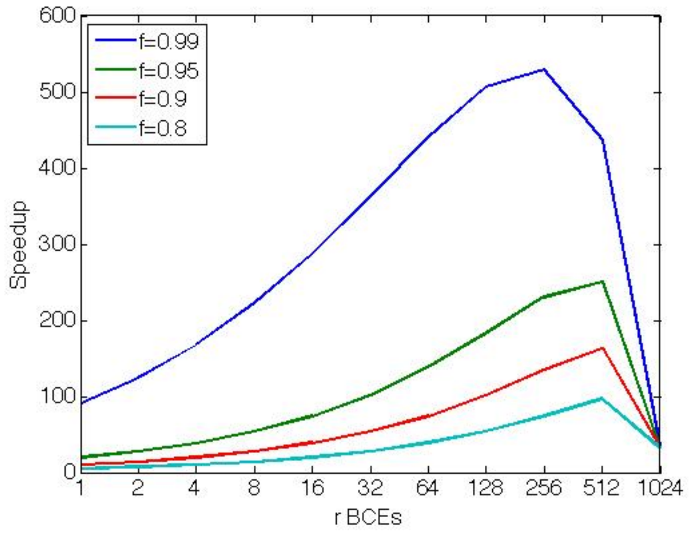


Figure 2.5: Amdahl's law in multi-core with asymmetric configuration by assigning r-BCE core for serial section and remaining n-r 1-BCE cores for parallel section with n=1024.

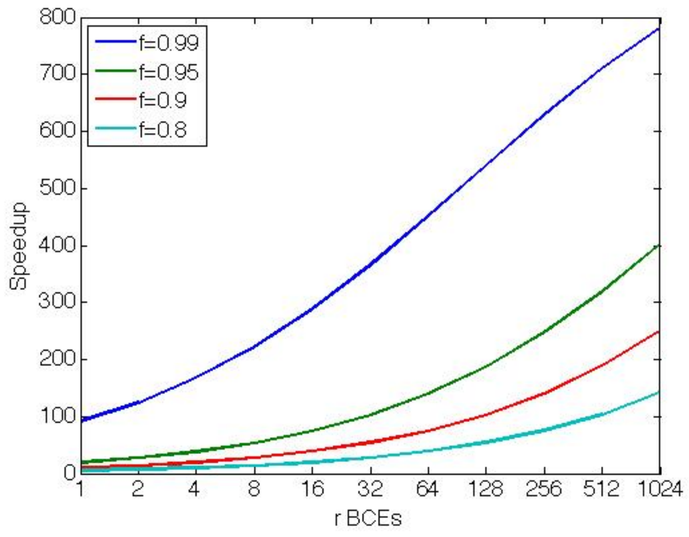


Figure 2.6: Amdahl's law in multi-core with dynamic configuration by varying r from n BCE to 1 BCE with n=1024.

(i.e., multiple threads wanting to enter the same critical section). They also considered same 3 different hardware configuration like Hill *et al* and arrived on slightly different conclusions compared to Hill *et al.* 1) Asymmetric multi-core processors deliver less performance benefits relative to symmetric processors, the reason being that contending critical sections are executed on the smaller cores rather than on the relatively larger cores in asymmetric multi-core processor, 2) Their findings advocates to use medium to larger symmetric cores in contrary to many tiny cores as inferred by Hill *et al* as the relatively larger small cores will speedup the CS, 3) In dynamic multi-core model, achievable speedup also depends on the accuracy of CS contention predictor which enables thread migration to bigger cores to execute the CS dynamically on the available big core. Their model targets the parallel application with homogeneous workload and assumes that critical sections are entered at random times as the goal of the model was not to present accurate performance numbers but to provide insight and intuitions for the future work.

Yavits *et al* [YMG14] also extended Amdahl's law by considering the effects of data exchange between the cores executing sequential and the parallel sections at the beginning and the end of each parallel section of a program (sequential-to-parallel synchronization) and data exchange among the cores executing the parallel section of a program (inter-core communication). They considered two main parameters in analyzing the speedup namely synchronization intensity and connectivity intensity. Synchronization intensity is the ratio of the number of data elements transferred during sequential-to-parallel data synchronization to the number of arithmetic operations. Connectivity intensity is the ratio of the number of data elements transferred during inter-core communication to the number of arithmetic operations. They conclude that, to improve the scalability and performance of a multi-core, it is important to address the synchronization and connectivity intensities of parallel algorithms as their parallelization factor f.

Madhavan *et al* [MJS11] analyzed the scalability of a set of data mining workloads that have non-negligible serial sections in the reduction phase of the program. They show that the reduction operation in such application grows linearly with number of cores by extending the Amdahl's speedup model. They incorporated the impact of reduction operations in the speedup of applications on symmetric as well as asymmetric CMP designs. Their analytical model estimates that asymmetric CMPs with one large and many tiny cores are only optimal for applications with a low reduction overhead. However, as the overhead starts to increase, the balance is shifted towards using fewer but more capable cores. This eventually limits the performance advantage of asymmetric over symmetric CMPs.

2.3.3.1 Limitations

Amdahl's law by itself is a pessimistic model which considers the serial section is constant and all the above mentioned variations tries to capture different elements from the parallel parts which contributes to the serial part in the multi-core era like critical section, inter-core communication, serialization in the reduction phase etc. These are probabilistic models and can help the researchers in analyzing the future trends on a very abstract level but can also lead to a faulty conclusion as they do not involve any real measurement of an application running in a given platform. This can be a bigger limitation for the application developers and processor architects as the accurate scalability will be harder to realize using probabilistic models and the methodology to over come this will be discussed in detail in Chapter 3 using our empirical Serial/Parallel Scaling (SPS) model.

2.3.4 Gustafson's Law for multi-core era

Juurlink *et al* [JM12] extend Gustafson's law to symmetric, asymmetric and dynamic multi-cores to predict multi-core performance. They claimed that neither the parallel fraction remains constant as assumed by the Amdahl's law nor it grows linearly as assumed by Gustafson's Law. Therefore, they proposed the Generalized Scaled Speedup Equation (GSSE) as shown in Eq. 2.7. GSSE is an intermediate model where the amount of work that can be parallelized is proportional to a scaling factor *Scale(P)*=$\sqrt{P}$, where P is the number of processing units employed. Their conclusion was that asymmetric and dynamic multi-cores can provide performance advantage over symmetric multi-cores. But again, the impact of serial part was neglected in this model .

$$speedup_{GSSE} = \frac{(1-f) + (f * Scale(P))}{(1-f) + \frac{(f*Scale(P))}{P}} \tag{2.7}$$

2.3.4.1 Limitations

Gustafson's law is based on very rough assumptions that do not correspond to the effective behavior of applications, and is over-optimistic. In particular, it assumes that the execution time of the serial part of the application is negligible and the achievable performance on the parallel part scales linearly with the number of processors and the workload size. In simple words, if we increase the workload of the program linearly and number of execution units linearly then the total execution time of the parallel program should remain constant. This is also a probabilistic model and these assumptions do not hold on real applications, even for limited thread number as there are various factors which can affect the achievable speedup [EDBE12].

2.3.5 The KILL Rule for Multi-core

KILL in kill rule for multi-cores [AL07] stands for *Kill if Less than Linear*, and represents a design approach in which any additional area allocated to a resource within a core, such as a cache, is carefully traded off against using the area for additional cores. The Kill Rule is a simple scientific way of making the tradeoff between increasing the number of cores or increasing the core size. The Kill Rule states that resource allocated to the core must be increased in area, only, if the core's performance improvement is at least proportional to the core's area increase. In another sense, increase resource size only if for every 1% increase in core area there is at least a 1% increase in core performance. This rule was clearly one of the guiding principles behind the development of the Tilera many-core chip.

2.4 Summary

In this chapter, we explored different ways in which performance can be estimated and measured. Their utilization varies according to the level of details modeled, complexity, accuracy and run time of the performance evaluation technique to obtain the results. Also, appropriate techniques should be used depending on the specific purpose of the evaluation as seen in Table 2.1. In order to extrapolate the performance of current parallel applications in the future many cores, simple models like Amdahl's law [Amd67] or Gustafson's law [Gus88] are often invoked. Amdahl's law: if one wants to achieve better response (improve peak performance); Gustafson's law: if one wants to provide better service (improve throughput). These law's have the merit to be very simple and to provide a rough idea of the achievable performance. But, they are very optimistic models for many-cores as they do not consider the impact of the serial section. We also looked into other enhanced probabilistic models which captures the effects of other performance limiting factors in the parallel parts of the application ([HM08], [EE10], [YMG14], [MJS11], [JM12]) such as synchronization, inter core communication, coherence etc.

In the next chapter, we will propose our Serial/parallel Scaling (SPS) model which overcomes the limitations faced by the probabilistic models. Using the empirical model of different applications, we show that the impact of serial scaling in MT application cannot be ignored in the many-core era. SPS model empirically captures the application behaviour in a given architecture as a function of Input set/problem size and number of processors. This can help the application scientists to understand the application-architecture interaction and to estimate the achievable speedup on scaling them to many-cores.

Chapter 3

The Serial/Parallel Scaling (SPS) performance model

In this chapter, we introduce the Serial/Parallel Scaling performance model which can be used to study the application scalability in the future many-cores. Then, we validate this model in two many-cores 1. Few large core Xeon Westmere processor with 24 logical cores and 2. Many small core Xeon-Phi processor with 240 logical cores. Finally, we discuss on what we infer from the model which will help in future many-core design.

Application developers require high level execution time model to understand the application scalability. Architects require such models to study the design space of the hardware to overcome the hardware bottleneck in future designs. Hence, there is a need for a performance model which can help both communities by revealing the application's inherent behaviour and the impact of the underlying architecture on the performance of the application.

Estimating the potential performance of parallel applications on the yet-to-be-designed future many cores is very speculative. The simple models proposed by Amdahl's law [Amd67] (fixed input problem size) or Gustafson's law [Gus88] (fixed number of cores) do not completely capture the scaling behaviour of a multi-threaded (MT) application leading to over estimation of performance in the many-core era. On the other hand, modeling many-core by simulation is too slow to study the applications performance.

In this chapter, we propose a more refined but still tractable, high level empirical performance model for multi-threaded applications, the Serial/Parallel Scaling (SPS) Model to study the scalability and performance of applications in many-core era. SPS model learns the application behavior on a given architecture and provides realistic estimates of the performance in future many-cores. Considering both input problem size and the number of cores in modeling, SPS model can help in making high level decisions on the design choice of future many-core applications and architecture. We validate the model on the few large core many-core Intel Xeon with 24 cores and many small core many-core processor Many-Integrated Cores (MIC) Xeon-phi with 240 logical cores.

3.1 What is sequential or serial part in a multi-threaded program?

Previously proposed execution time models as well as our SPS model assumes that the execution of an application can be arbitrarily split into a serial part and a parallel part. *The serial part is constituted of the sections where only a thread is actively running. The parallel part consists of the sections where several threads can run concurrently.* This is illustrated in the Figure 3.1 for Delaunay triangulation program from LONESTAR benchmark suite between time samples 445-480, 493-530 and 537-541ms and in Figure 3.2 for Bodytrack program from PARSEC benchmark suite between 25-30, 60-65 and 100-105ms.

In a multi-threaded program, three contributions to the sequential part can be discriminated at a very high granularity. First, the code executed by the main thread of the program before the threads are spawned and the final code executed after they are joined. Second, after the parallel threads are spawned, the master or main thread may have to execute some serial work to manage the worker thread pool or to execute some code after a global synchronization. This part of the application is often referred to as Region Of Interest (ROI). Third, critical sections executed in a parallel section may cause serial execution of the parallel threads.

To capture serial and parallel sections in the execution of a parallel application, we use coarse grain monitoring of the threads and a heuristic to classify threads as active or inactive. Since critical section execution is generally quite short, the execution of critical sections will be generally classified in the parallel part of the application and are captured as a parallel performance limiting factor in this thesis.

3.2 Motivation and Related Work

Two simple models Amdahl's law [Amd67] and Gustafson's law [Gus88] are still widely used to extrapolate the theoretical performance of a parallel application on a large machine. They correspond to two very different views of the parallel execution of an application. We will refer to these two views as the fixed workload perspective and the scaled workload perspective respectively(illustrated in Fig. 2.1).

Fixed workload perspective Amdahl's law assumes that the input set size (workload) of an application remains fixed for a particular execution. The objective of the user is to reduce the computation time through executing the program on a parallel

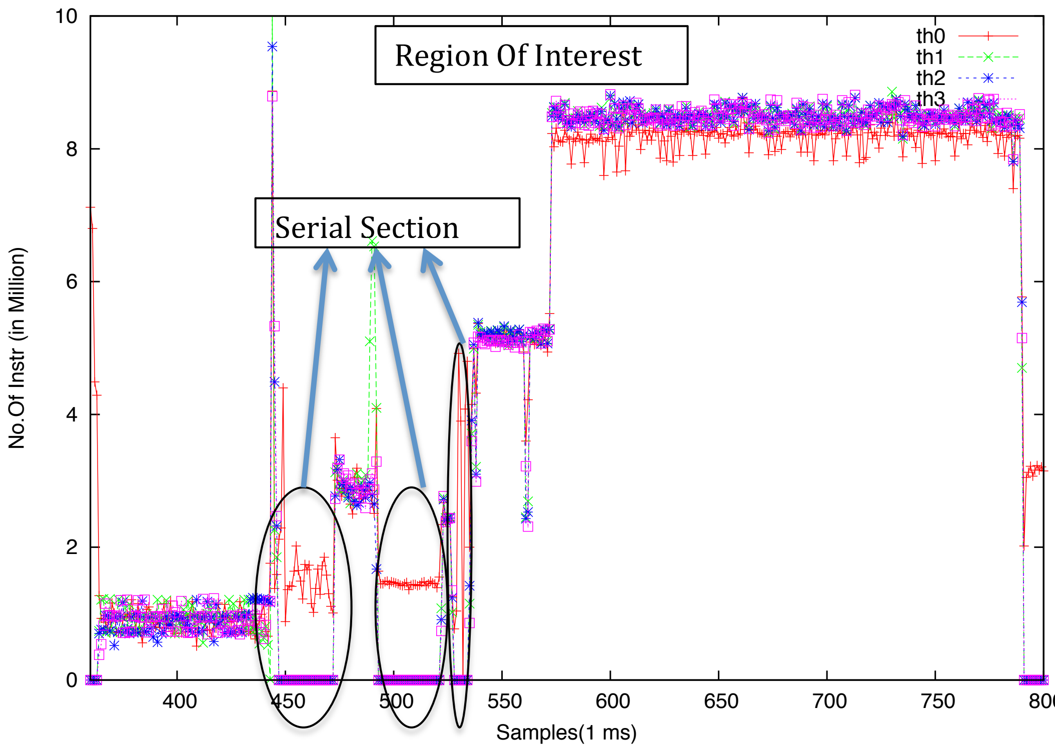


Figure 3.1: Deltri showing the existence of serial section in the ROI.

hardware. *This perspective assumes that the fraction of serial part in a program remains constant for any input set size.*

Scaled workload perspective Gustafson's law assumes implicitly a very different scheme for parallel execution. The objective of the user is to resolve the largest possible problem in a constant time. *This perspective assumes that the relative part of the parallel computation grows with the problem or input set size but ignores the serial section.*

There are few analytical models that were proposed in the past which looks at the performance analysis of an application in multi-core systems by extending the Amdahl's and Gustafson's law. In [JM12], Juurlink *et al* extend Gustafson's law to symmetric, asymmetric and dynamic multicores to predict multicore performance. They claim that neither the parallel fraction remains constant as assumed by the Amdahl's law nor it grows linearly as assumed by Gustafson's Law and proposed a Generalized Scaled Speedup model with parallel scaling factor $Scale(P)=\sqrt{P}$. Further, extending the Amdahl's passive model, Hill *et al* [HM08] proposed a performance-area model called *Amdahl's law in the multicore era*. Eyerman *et al* [EE10] introduced a probabilistic model which shows that, even the Critical Section(CS) in the parallel part contributes

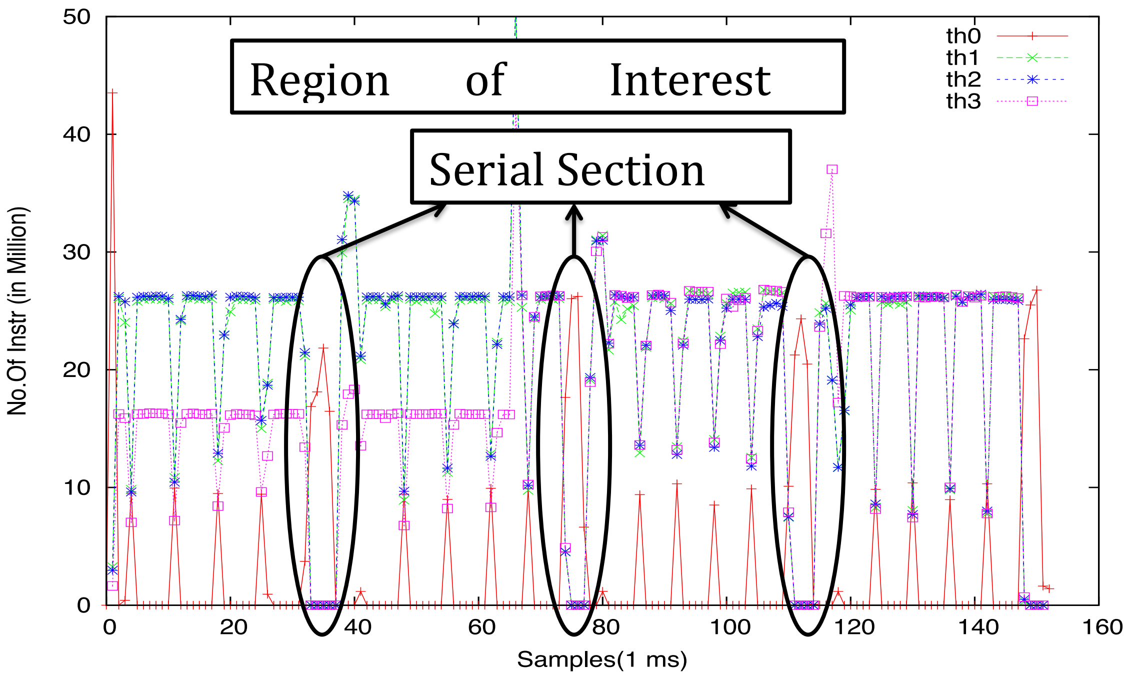


Figure 3.2: Bodytrack showing the existence of serial section in the ROI.

to the serial section of the program. Yavits *et al* [YMG14] also extended Amdahl's law by considering the effects of sequential-to-parallel synchronization and inter-core communication. We have discussed all these models in detail in Chapter 2.

Existing performance models are too generic as they neither consider application behaviour nor the impact of the underlying architecture. They have the merit to be very simple and provide a rough idea of the possible performance. But they are very optimistic models. For some applications, the execution time of the serial-section increases significantly with the increase in input size, but also at times slightly with the increase in number of processors. Fig. 3.3 shows four different serial scaling behaviours on different applications when the input set size (I) is increased.

1. Both serial and parallel execution time grow at different rates with I. Eg. *Bodytrack*.

2. Both serial and parallel execution time grows linearly with I. Eg. *Deltri*.

3. Serial section is ignorable and independent of I. Eg. *Swaptions*.

4. Serial section is not ignorable and independent of I. Eg. *Canneal.*

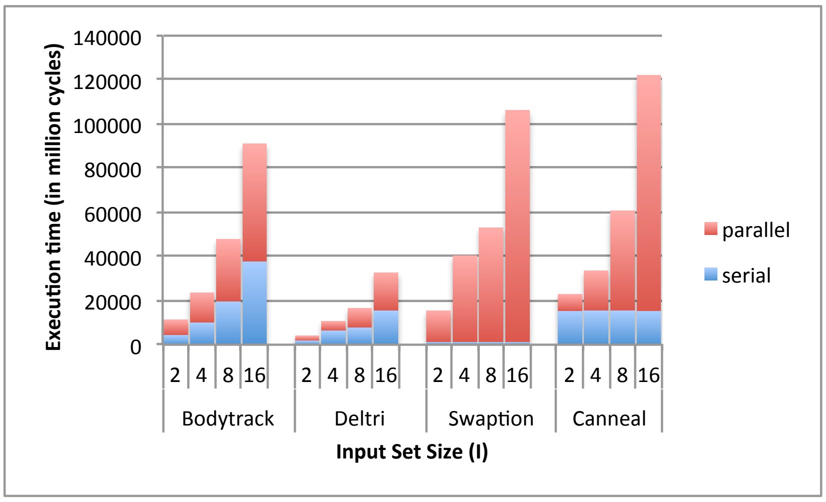


Figure 3.3: Different application scaling behaviour with variation in input set size.

The reason for different serial scaling behaviour among applications can be attributed to the parallelization technique. Multi-threaded programs generally have 3 major phases. 1) the initialization phase where input data are generated, 2) the Region Of Interest (ROI) where the main computation is executed and 3) the finalization phase where the results are processed and the program is terminated. Initialization, finalization phase belongs to the serial part and the ROI can belong to both serial and parallel parts depending on the parallelization technique used. In data parallel application, once the threads are spawned they work until the assigned job is complete without any intervention. Here, ROI is totally parallel. This behaviour is observed in *swaptions, canneal.* On the other hand, the applications that uses pipeline parallelism or a worker thread pool based implementation has a ROI which contributes to the serial part. Here, the master thread does some work to feed the worker threads in ROI and can be a significant contribution to serial section and scales with Input set size. This behaviour is observed in *bodytrack, deltri.*

We build on the observation that not all applications are scaling the same way with the number of processors and the input set size. We use monitoring and measurement techniques to obtain the experimental data for different applications. Then, apply non-linear regression to fit the model and obtain the scaling parameters. In the past, there are few works that explored regression and machine learning based approaches to predict the application performance on a given multi-processor platform. Bradely *et al* [BRL$^+$08] explored regression based approach to predict parallel program scalability in large scale scientific applications in supercomputers. Their main goal was to help the scientist know the scalability of their applications, knowing which, the worthless total demand on the system can be reduced and the efficiency of the application can be improved. They achieved low error model by separately measuring computation and communication times. In [IdSSM05], authors used multilayer neural networks (black-box approach) to train the input data from executions on the target platform to predict the performance of parallel program in large system. They created an empirical model for SMG2000 parallel application and were able to predict performance within 5-7% error range across large and multidimentional parameter space. In [LBdS$^+$07], authors used statistical techniques such as clustering, association, and correlation analysis, to infer the behavior of application parameters, then, compared two different predictive models built using piecewise polynomial regression(splines) and artifical neural networks. They concluded that the accuracy of neural networks was on par with spline-based regression models, but, compared to neural networks, spline-based regression models provide more insight. In our work, the considerations are bit different as we target a many-core in a single die where communication cost will be highly reduced and the serial section may become a real bottleneck. So, we apply the regression technique on the serial and parallel measured data to obtain a high-level empirical performance model that can be used to understand the impact of serial scaling of the multi-threaded applications in the future many-cores.

In next section, we present the SPS model that empirically captures application scalability as a function of Input set/problem size and of number of processors, and thereby provides realistic extrapolation of application performance in the many-core era.

3.3 The SPS Performance Model

Our model's main objective is to extrapolate the multicore execution behavior of a parallel program to the future many-cores. To keep the model simple, we consider the following:

#1. The execution time is dependent only on input set size I and the number of processors/cores P i,e $t(I, P)$

#2. An uniform parallel section and an uniform serial section, i.e, we model the total execution time as the sum of serial and parallel execution times as shown in Eq. 3.1. Both execution times $t_{seq}(I, P)$ and $t_{par}(I, P)$ are complex functions,

$$t(I, P) = t_{seq}(I, P) + t_{par}(I, P) \tag{3.1}$$

#3. For both the execution times, the scaling with the input set size (I) and the scaling with the number of processors (P) are independent i,e. t_{seq} and t_{par} can be modeled as: $t_{par}(I, P) = F_{par}(I) * G_{par}(P)$ and $t_{seq}(I, P) = F_{seq}(I) * G_{seq}(P)$. General observation is that, the execution time of an application with constant input set size reduces with number of threads and the execution time increases gradually when input set size is increased with fixed number of threads. Linear equations do not satisfy the trend and hence, we are using a non-linear power model such that F and G can be represented by a function of the form $h(x) = x^{\alpha}$. Thus, the general form of execution time of the parallel execution is:

$$t(I,P) = c_{seq} I^{ISS} P^{PSS} + c_{par} I^{IPS} P^{PPS} \tag{3.2}$$

The SPS model only uses 6 parameters which are obtained empirically to represent the execution time of a parallel application, taking into account its input set and the number of processors. c_{seq} , ISS and PSS are used to model the serial execution time and c_{par}, IPS and PPS are used to model the parallel execution time[1]. c_{seq} and c_{par} are serial and parallel section constants which gives the initial magnitude of the execution time. ISS and PSS are the *Input Serial Scaling* parameter and the *Input Parallel Scaling* parameter. IPS and PPS are the *Processor Serial Scaling* parameter and the *Processor Parallel Scaling* parameter.

In particular, Amdahl's law and Gustafson's law can be viewed as two particular cases of the SPS model.

A comparison with Amdahl's Law Amdahl's law assumes a constant input I_{base} and an execution time of the serial part independent from the processor number, i.e. $PSS = 0$. It also assumes linear speedup with the number of processors on the parallel part, i.e $PPS = -1$. Substituting the values in Eq. 3.2, we get Eq. 3.3 which shows that execution time is dependent only on P.

$$t(I,P) = c_{seq} I_{base} + \frac{c_{par} I_{base}}{P} \tag{3.3}$$

A comparison with Gustafson's Law Gustafson's law assumes constant execution time for the serial part, i.e. independent of the working set ($ISS = 0$) and the number of processors ($PSS = 0$). Therefore, $t_{ser}(I,P) = c_{seq}$. It also assumes that the input is scaled such that 1) the parallel workload I_{Gus} executed with P processors is equal to P times the "parallel" workload executed in one processor, i.e., $I_{Gus}^{IPS} = P$. 2) speedup on the parallel part is linear, i.e. $PPS = -1$. Substituting the values in Eq. 3.2, we get Eq. 3.4 which shows that time taken to execute remains constant.

$$t(I,P) = t(P * I_{base}, P) = c_{seq} + c_{par} \tag{3.4}$$

SPS model can be used to extrapolate the execution time for extended version of the same architecture to study the over all application scalability but cannot be used to find and resolve the hardware bottleneck[2] in the system.

In the next section we present the methodology we adopted to empirically determine the 6 parameters of the SPS model.

[1]As the per core parallel execution time includes the time spent in synchronization and core-core communication, our model for the parallel part accounts them by default

[2]Scalability of the application can be poor due to resource contention or insufficient resources.

3.4 Experimental Methodology

3.4.1 Platform

We ran our experiments on currently available few large many-core Intel Xeon Westmere processor E5645 and many small many-core Intel Xeon-Phi 5110P system. The configuration of the system are shown in Table 3.1.

Table 3.1: Experimental system specification.

Processor	Intel Xeon E5645	Xeon-Phi 5110P
# of Cores	6 cores x 2 sockets	60
# of Threads	24	240
Clock speed	2.4 GHz	1.053 GHz
Cache	12 MB	30 MB
Memory	32 GB	8 GB

3.4.2 Benchmarks

In this study, we focused on applications that will be executed on future manycores. Therefore, we consider benchmarks which are parallelized with shared memory model using Pthreads library. The two conditions that were necessary for our experiments are: 1) Program should be able to run from 2 to 24 (resp. 240) threads and should be load balanced. 2) Input sets had to be generated with known scaling factors. We investigated two different categories of benchmark suites as our case study. They are 1) Regular parallel programs from the PARSEC benchmark suite and 2) Irregular parallel programs from the LONESTAR benchmark suite.

3.4.2.1 Regular Parallel programs

This class of applications operate on arrays and matrices where the data can be clearly partitioned and can be processed over multiple cores in parallel. We chose the PARSEC benchmark suite [BKSL08] for our study of regular programs. We studied Bodytrack[3], Canneal, Fluidanimate and Swaptions in PARSEC. The reason for not considering other benchmarks are as follows: Blackscholes and Facesim have constant input set size (doesn't have input set size scaling parameter). Streamcluster, Ferret, Dedup spawns extra threads which exceeds the processor resource when scaled linearly[4]and

[3]Verified in Xeon architecture where we were able to build it but not build-able in Xeon-phi.

[4]2N, 4N+3, 3N+1 respectively.

x264 spawns constant 32 threads. Freqmine and Vips were not build-able in the given platform.

Most of the PARSEC benchmarks are data parallelized or pipeline data parallelized where load balancing is usually maintained. Bodytrack is a computer vision application which tracks the human movement by processing the input frame by frame. It implements a worker thread pool where the main thread works like a master thread and the other threads are worker threads which can be observed in Figure 3.2. In this application, we can notice that with the increase in the input set size, the serial section increases linearly (Figure 3.3). From source code analysis, we understand that these sections are used to downscale the image before the output creation process and does some useful computation.

3.4.2.2 Irregular Parallel programs

Irregular programs operate over pointer-based data structures such as trees and graphs. The connectivity in the graph and tree makes the processing data dependent and memory access pattern are unknown before the input graphs are known. Due to this reason, static compiler analysis fails to unveil total parallelism available in such applications. The LONESTAR [KBCP09] benchmark suite consists of irregular programs which use the Galois run-time system [KPW^{+}07], [KBI^{+}09] to exploit the underlying parallelism.

In LONESTAR benchmark suite, we studied *Delaunay triangulation (deltri), preflowpush (preflow), Single-Source Shortest Path (sssp), Boruvka's Algorithm (Bourvka), barneshut (barnes), Survey propagation (survey).* We noticed the serial scaling behavior is prominent in *deltri, preflow and survey.* In deltri, serial part is mainly contributed by adding the points to the worklist and dividing the work. In *preflow, survey* the serial part is due to the local graph computation which builds the customized graph for further parallelization. On analyzing these benchmarks, we observed that the serial part of the program scales with the input size in the ROI.

3.4.3 Input set scaling

PARSEC benchmark input sets have linear component scaling parameter [BL10] which are used to scale the input set. Similarly, for LONESTAR we can generate the mesh and graphs with linearly increasing nodes. The base input set size chosen for the benchmarks are shown in Table 4.1. The input is scaled by multiplying the base size with the scaling factor like 2,4,8 etc for different experimental runs. For example, for $I = 4$ for *fluidanimate*, we used $Frames = 4$.

3.4.4 Methodology

We used a 2 step approach to obtain the 6 SPS model parameters as described below;

Step 1 - Data collection: We collected the Performance monitoring Unit (PMU) samples (number of instructions executed , number of unhalted clock cycles) using tip-

Benchmarks	Input type	Base Size Xeon	Base Size Xeon-Phi
Fluidanimate	Frames	2	2
Canneal	Swaps per step , swaps	100K, 64	100K, 64
Swaptions	Swaptions, simulations	32, 100K	32, 100K
Bodytrack	Frames	16	4
Survey Propagation	Variables, Clauses	400K,3	400K,3
Deltri	Mesh	262144	131072
Preflow, SSSP, Bourvka	Random graph	2097152	1048576
Barneshut	Bodies	16384	2048

Table 3.2: Base input set

top [Roh11] at a regular interval of 1ms. Tiptop works on unmodified benchmarks and does not require code instrumentation. The events are counted on per thread basis[5]. Our thread spwaning strategy was different for both Xeon[6] and Xeon-Phi[7]. The thread wise activity of the application is analyzed and the execution time spent in the serial and parallel parts are calculated from the number of unhalted clock cycle event. Determining whether the execution is serial or parallel is done empirically: on a given 1ms time slice, a thread is classified as active if it exceeds a minimum CPU utilization threshold ($> 1\%$). For example, in Fig. 3.2, *bodytrack* benchmark is illustrated. th0 is the master thread and th1, th2, th3 are worker threads. We observe that in between the time samples 25-30, 60-65, 100-105 (approximately), the parallel threads are inactive and the master thread is active, thus contributing to the serial section.

[5]Proper measures were taken to avoid randomness due to context switching by pinning the threads to the logical cores

[6] In Xeon, we distributed our threads among the sockets and are aware that socket communications are expensive. We populated one thread per core initially, after the 12th thread we used hyper threads in the cores.

[7]We spawned all threads in single core first then moved to next core. In Xeon-Phi for the core to be busy it should have at least 2 threads as the instruction are fetched across 4 way SMT in a round robin fashion.

Step 2 - Modeling: The above mentioned steps were performed for every application on the given hardware by varying the number of threads(P) and the input set size(I) which is further explained in section. 3.4.5 and execution time $t_{seq}(I, P)$ and $t_{par}(I, P)$ are obtained. Then, we perform a regression analysis with the least-square method to determine the best suitable parameters for the available experimental data.

3.4.5 Validation

To validate the model, we use holdout cross-validation method [LÖ09]. We divide our obtained data into *trainingset* and *validationset*. Training set is a data subset $(I \leq 16, P \leq 16)$ which is used to tune the model to obtain its parameter values with non-linear regression and validation set is the data subset on which the models prediction capability on the given architecture will be validated. As our model is based on $t(I, P)$, our data set contains execution time (in million cycles) for the application with given I and P.

$$\%error = \frac{MeasuredValue - PredictedValue}{MeasuredValue} * 100 \tag{3.5}$$

If the % error is positive, then the execution time is under predicted and if negative, the execution time is over predicted. We validate[8] our model with the validation set $\{I{=}32, P \leq 240\}$ except for fluidanimate where $P = 128$ as it spawn threads in the order of 2^n. We show the prediction error range of every application using a box-plot which shows the minimum, maximum and median percentage errors in these applications. We can observe from Figure 3.4 that the median error for predicting the application performance running 240 threads are in the range of 3 to -20 percentage which can be acceptable for our high level modeling as SPS model gives upper bound of the achievable performance. Figure 3.5 shows the how well the model fits the measured value for *preflow*. Error percentage can be further minimized by using more training samples or by using resource aware modeling of the parallel section by knowing the applications per thread memory access pattern and communication pattern with other cores which is a potential future work.

To show the goodness of fit statistically, we found the absolute correlation (R-Squared) between observed and predicted values as shown in Eq. 3.6, where, y_i is the observed value, $\hat{y_i}$ is the predicted value of the i^{th} sample in the test set and $\bar{y}$ is the mean of the samples in test set. R^2 is a statistical measure that shows how close the experimentally obtained data are to the fitted regression line. Higher the R^2, better the model fits the data. In our model for different applications under study, R^2 values are high in the range $0.9945 \leq R^2 \leq 0.9999$ which means that the predicted value is almost equal to observed value and data points would fall on the fitted regression line.

[8]Only validation for Xeon-Phi many-core is shown in this chapter. The model parameters as well as validation for Xeon Westmere is shown in the appendix. This keeps the chapter simple and makes it easier for the readers to understand the model parameters better.

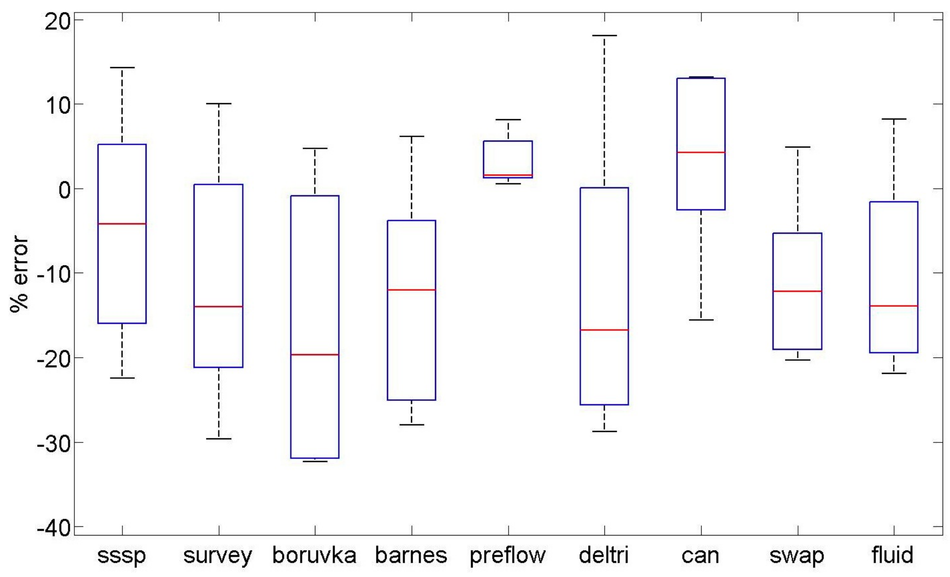


Figure 3.4: Box-plot showing the MIN, MEDIAN, MAX error in predicting $\{I = 32, P \leq 240\}$ for different applications.

$$R^2(y, \hat{y}) = 1 - \frac{\sum_i (y_i - \hat{y}_i)^2}{\sum_i (y_i - \bar{y})^2} \tag{3.6}$$

3.4.6 Analysis of SPS model parameters

Table. 3.3 reports the SPS model parameters that were computed from our experiments on Xeon-Phi. Applications have different behaviors as illustrated by the parameters. From the model parameters, we can understand the scaling pattern of the serial and parallel sections of the application in a given architecture. ISS and IPS will be positive and the value explains the impact of input scaling in the execution time. For example, low ISS values of *canneal, fluidanimate* in serial section shows that the impact on serial scaling due to input set size is low and constant. On the other hand, applications like *survey*, *preflow*, *deltri*, *survey* and *boruvka* have quite significant serial sections which grows quasi-linearly with the input set size and serial scaling can't be ignored. For most applications, ISS and IPS are in the same range, i.e. serial section scales similar to parallel section while for *barneshut* the execution time of the parallel section grows quadratically faster than the serial section as the input set size is scaled.

Serial section by definition will run on a single processor. Therefore, changing the number of cores to execute the application should not affect the serial execution time. We can observe from the model that PSS is either positive or negative but its value is low and negligible.

PPS will be negative, as it shows the reduction in execution time of the parallel part

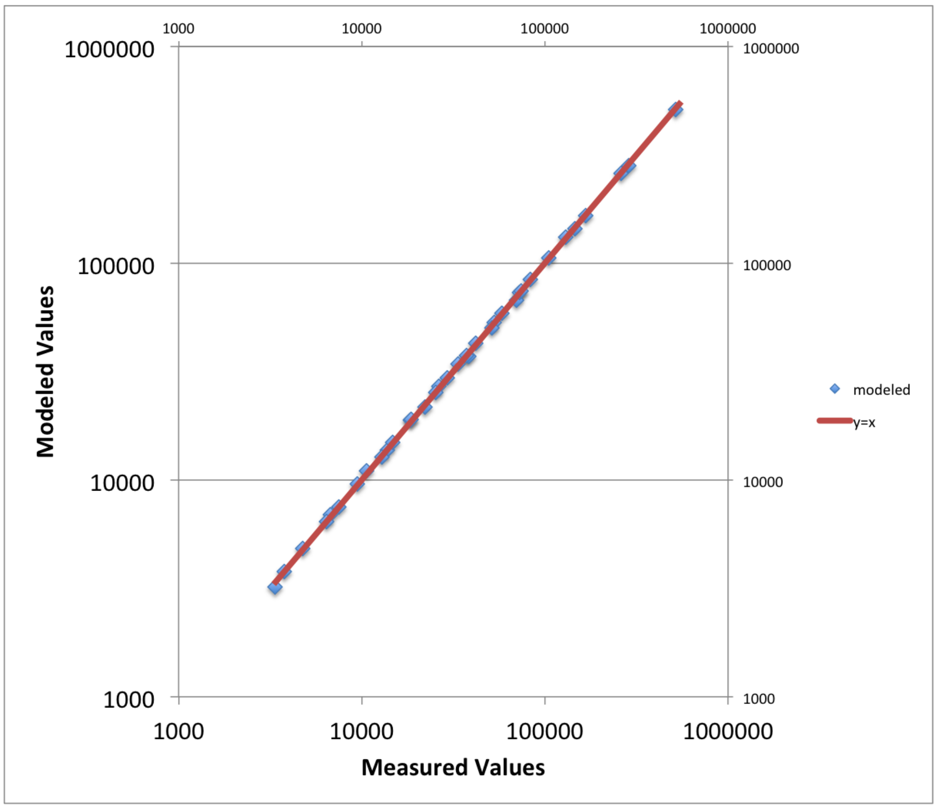


Figure 3.5: Graph showing the measured value versus modeled value for $preflow$ along varying input set size and number of cores.

	serial section	Parallel section
can	$14725.89I^{0.001}P^{0.003}$	$32138.1I^{0.95}P^{-0.873}$
swap	0	$33367.4I^{1.035}P^{-0.744}$
fluid	$1163.46I^{0.002}P^{0.076}$	$6438.618I^{1.024}P^{-0.783}$
deltri	$2716.98I^{0.994}P^{-0.007}$	$72750.5I^{1.03}P^{-0.933}$
preflow	$1334.87I^{0.965}P^{-0.001}$	$103915.3I^{0.978}P^{-0.979}$
boruvka	$492.76I^{0.978}P^{-0.023}$	$27935.0I^{1.061}P^{-0.709}$
barneshut	$10.078I^{1.004}P^{-0.027}$	$593.015I^{2.119}P^{-0.896}$
sssp	$50.24I^{0.975}P^{-0.031}$	$4528.222I^{1.002}P^{-0.76}$
survey	$729I^{1.01}P^{-0.024}$	$60685.8I^{1.204}P^{-0.978}$

Table 3.3: SPS parameters for different applications executed on Xeon-Phi obtained using trainingset$\{I \leq 16, P \leq 16\}$

with the increase in number of processors executing the application. If $PPS \approx -1$, it implies that the parallel section of the application scales well on the given architecture. Among the chosen applications only $deltri, preflow, survey$ has good parallel scaling. Rest all benchmarks have average parallel section scaling i,e. $PPS \geq -0.9$.

In next section, we discuss the inference of these application specific models on Xeon-phi and contrast SPS model with the other existing models.

3.5 Inference

In this section, we explain the inference of the observation using our model and also show how the serial section impacts the speedups of the application with varying I and P.

3.5.1 The f parameter

SPS model allows to overcome a major difficulty with Amdahl's and Gustafson's laws: the identification of the parameter f which is usually assumed. With our model, we can find f empirically using Eq. 3.7. Fraction of parallel part (f) in a program varies with I according to our model. In Eq. 3.7, we can see that f is basically a function of I (I_{base} is a constant base input set size).

$$f = \frac{t_{par(I_{base},1)}}{t_{ser(I_{base},1)} + t_{par(I_{base},1)}} = \frac{c_{par} I_{base}^{IPS}}{c_{seq} I_{base}^{ISS} + c_{par} I_{base}^{IPS}} \tag{3.7}$$

Variation in f for different application are captured in Table. 3.4 by varying Input set size (I) from 1 to 10000. We can infer the following:

1. *Completely parallel application:* Some applications are completely parallel i,e $f = 1$ the way Gustafson's law assumes for example *swaptions.*

2. *Constant serial section:* In *canneal and fluidanimate* the serial part is independent of I or constant as we can notice from the parameters of SPS model in Table 3.3. Parallel fraction f improves with I. In such applications, the larger parallel scaling amortize the constant serial section. Therefore, larger the input set size, larger the parallel fraction f in the program.

3. *Sub-linear or linear serial section:* The impact of the serial scaling can be noticed in *deltri, preflow, boruvka, sssp, survey.* In these applications the serial part grows sub-linearly or linearly with the parallel part when we increase the input set size. Therefore, the parallel fraction improves very marginally or remains almost the same though we increase input set size. Exception to this is *barneshut* as the parallel section grows quadratically compared to the serial part and the linear serial growth is amortized.

Benchmark	1	10	100	1000	10000
can	0.860	0.982	0.998	1.000	1.000
swap	1.000	1.000	1.000	1.000	1.000
fluid	0.847	0.983	0.998	1.000	1.000
deltri	0.959	0.966	0.972	0.977	0.981
preflow	0.985	0.986	0.986	0.987	0.987
boruvka	0.983	0.986	0.988	0.990	0.992
barneshut	0.983	0.999	1.000	1.000	1.000
sssp	0.989	0.990	0.990	0.991	0.991
survey	0.988	0.992	0.995	0.997	0.998

Table 3.4: Parallel fraction f for varying Input set size from I=1 to 10000 for applications executed on Xeon-phi.

Calculated f values show that, *the parallel fraction of an application is not constant as assumed by Amdahl's and Gustafson's law but varies with I.* This drawback is further illustrated in the comparative study in section 3.5.3.

3.5.2 Sub-linear scaling

The SPS model takes into account that the potential speed-up on the parallel section is sub-linear i.e., $PPS > -1$ in most of the benchmarks. Few benchmarks like *survey, preflow and deltri* have a good parallel scaling with $-1 \leq PPS \leq -0.9$ which means that their speedup can still be in between 1024 to 512 for a processor with 1024 cores. Large number of benchmarks have sublinear scaling in the range $-0.9 \leq PPS \leq -0.7$, e.g. *canneal, fluidanimate, swaptions, boruvka, sssp and barneshut* where the maximum achievable speedup will be in the range $512 \leq speedup \leq 128$ in a 1024 core machine. Added to the sub-linear parallel scaling, SPS model also captures the serial scaling effect with ISS and IPS hence comparatively lower speedup is achieved in reality.

Figure 3.6 illustrates the potential speedups extrapolated for a few benchmarks varying the processor number from 1 to 1,024 and varying the problem size from 1 to 10,000. The illustrated examples are representative of the behaviours that were encountered among the chosen applications. We discuss some of the interesting cases which gives better inference of the SPS model parameters and the sub-linear scaling behaviour of the applications.

1. Some applications are completely parallel. In this case, the speedup is totally dependent on how the parallel section of the application scales IPS and PPS. In

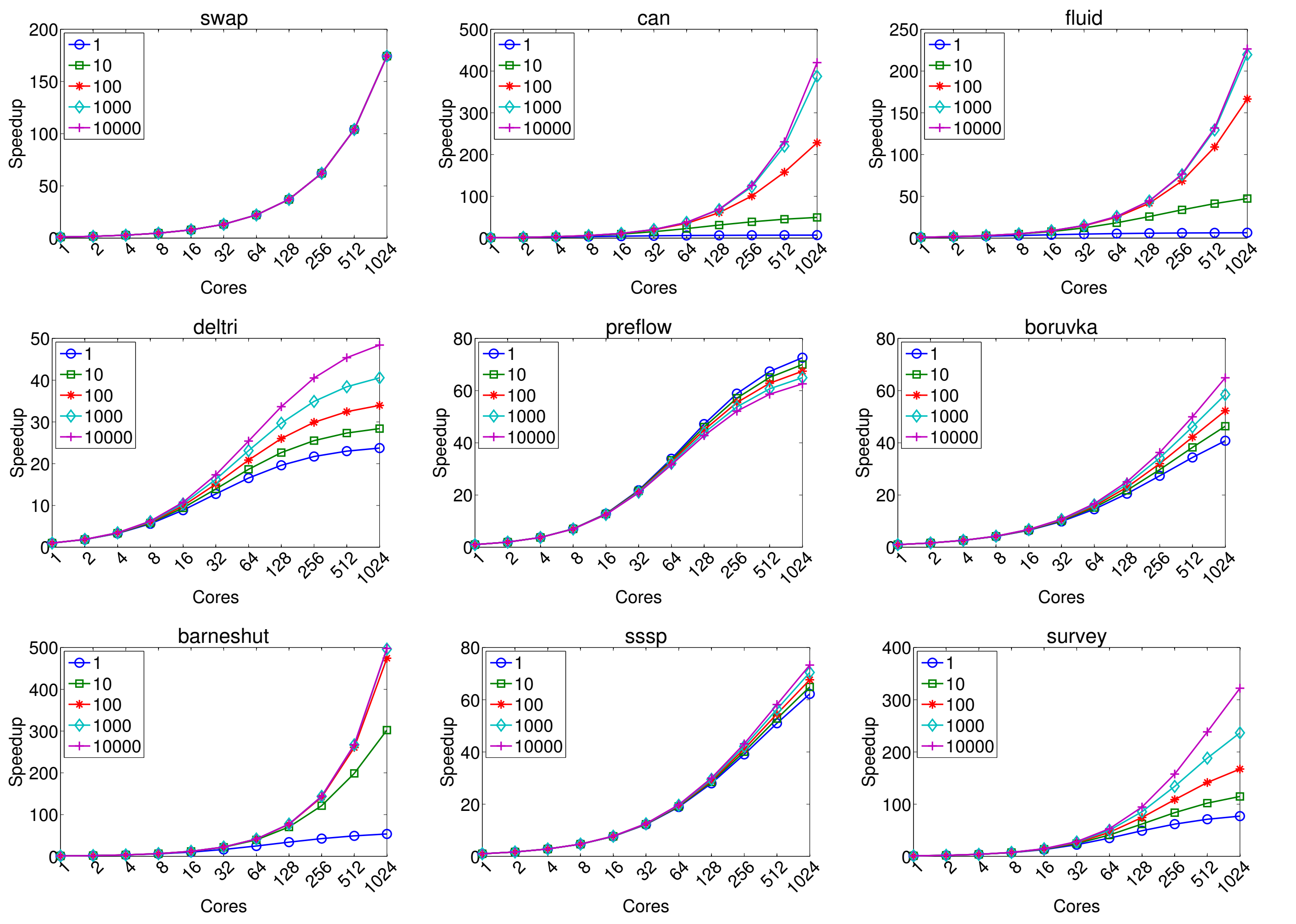


Figure 3.6: Potential speedups(Y-axis) extrapolated by varying processor number from P=1 to 1,024 (X-axis in log scale) and Input set size from I=1 to 10,000.

swaptions, the serial section is so small that it can be ignored and the parallel section has sub-linear scaling i,e $PPS = -0.744$, which limits the speedup to 174.

2. Some applications have almost constant serial part and good acceleration on the parallel part for every input set size. But, large input set sizes are needed to amortize the constant serial part which can be deduced directly from the parameters of the applications. In *canneal and fluidanimate* , large $\frac{c_p}{c_s}$, small ISS and PSS makes the serial section independent of I and P but the parallel section scales quasi linearly with I and P. Hence, we can achieve significant improvement in speedup using larger I. We can observe that the speedup increases with I for canneal and fluidanimate in Fig.3.6

3. In certain applications, serial part scales similar to the parallel part i,e $ISS \approx$

IPS and PSS is sublinear. We can notice such pattern in *deltri, preflow, survey, sssp, boruvka.* These kind of applications seldom benefit from a manycore system because their speedup gets saturated with P despite increasing I. In *deltri and preflow* we can notice that the saturation is very prominent. *SSSP and boruvka* has limited speedup because of the poor parallel scaling parameter $PPS \approx 0.7$ but will eventually get saturated if we increase P irrespective of I.

4. Even when the execution time of the serial section is increasing linearly with the input set, it does not always affect the scalability of the application. For instance, in *barneshut*, the execution of the serial section is also increasing with the input set size ($ISS = 1.004$) but at a much lower rate than the execution time of the parallel section ($IPS = 2.1$). Hence, the linear speedup can be noticed in *barneshut.*

3.5.3 Comparison with previous works

Previous models like Amdhal's, Gustafson's and GSSE were not empirical[9] and also were not considering the serial scaling and hence for some applications we find a very large difference in speedup when compared to SPS model. Our model gives more realistic speedups for every application because it considers the parallel and serial scaling parameters obtained by executing the application in given architecture. We show the comparison of speedup obtained using these models in Table 3.5 for all the applications if executed with $P = 1024$ and $I = 100$. f can be obtained from Eq. 3.7 and can be used in Eq. 2.1, Eq. 2.2, Eq. 2.7 to obtain Amdahl's, Gustafson's and GSSE speedup respectively, GSSE uses $\sqrt{p}$ as scaling parallel factor. SPS model speed up is calculated from $\frac{T(100,1)}{T(100,1024)}$.

We can observe that though the application is completely parallel the achievable speedup can be very low. For example, *canneal* has $f = 1$ and still has speedup of 174 which is mainly due to the poor parallel scaling which was not considered by other parallel models. While *Barneshut* comparatively has higher speedup with $f = 1$ because its parallel section scales quadratically and the serial section scales linearly with I. In applications like *deltri, preflow, survey, SSSP, boruvka* SPS model speedup is comparable with Amdhals as $ISS \approx IPS$ and it is slightly low due to the parallel scaling factor considered by SPS. Therefore, SPS model provides a realistic speedup considering the application behavior in the underlying architecture.

[9]Basically, to calculate the speed up for other probabilistic models we need serial $1 - f$ and parallel fraction f in the program. According to our approach, the fraction f needs to be obtained experimentally by measuring the runtime. We compute the f using Eq 3.7 and substitute in respective model.

Benchmark	f	Amdhal	Gustafson	GSSE	SPS
can	0.998	332.46	1021.92	961.39	228.15
swap	1.000	1024.00	1024.00	1024.00	174.25
fluid	0.998	386.53	1022.35	973.74	166.47
deltri	0.972	34.60	995.41	533.77	33.95
preflow	0.986	68.12	1009.97	708.92	67.42
boruvka	0.988	77.74	1011.83	739.46	52.28
barneshut	1.000	928.87	1023.90	1020.73	474.39
sssp	0.990	93.96	1014.10	780.35	67.64
survey	0.995	171.35	1019.02	885.63	167.16

Table 3.5: Achievable speedup for the applications with $I = 100$ and $P = 1024$.

3.5.4 SPS model limitations

We have developed the SPS model in order to extrapolate the performance of future parallel applications on large scale many cores featuring 100's or even 1000's of cores. It is our belief that future applications will in some way exhibit scaling characteristics within the same spectrum as the benchmarks we studied in this paper.

However, the SPS model remains very rough and should be used very carefully when drawing definite conclusions on the scaling of a given application on a many core. For example, the SPS model is able to predict that if the PPS is largely greater than -1 (e.g. $PPS = -0.75$) or if the serial section is scaling with the input set and the processor number. In such scenario the application is very unlikely to scale favorably with I and P. Even if the measured parallel scaling factor PPS is close to -1 and the serial scaling is very limited, a sudden limitation can appear due to some hardware bottleneck such as memory bandwidth, locks contentions, cache contentions ..etc and limit application scalability, which is not captured in the model.

Moreover, the inferred parameters are for a specific implementation of an architecture[10]. Changing the balance in the architecture -e.g. cache per processor ratio, bandwidth per processor- can change the scaling parameters of the application. Application re-engineering and algorithm modification will also change the scaling factors of an application.

[10]This can be observed with the speedup and f values for the Xeon Westmere system in the Appendix

3.5.5 SPS analysis rules

We list here a small set of rules of thumb to analyze the expected scaling behavior of a parallel application from its SPS model parameters collected on a small configuration and small working set.

- Maximum speed-up is limited by PPS: PPS close to -1 indicates that quasi-linear speedup is possible, PPS close to -0.5 limits the potential speed-up to the square root of the processor number.
- The serial section may scale with the input set size: if it scales with the same factor as the parallel section ($ISS \approx IPS$), the possible speedup will not increase with the problem size and will saturate soon.

3.6 Heterogeneous architecture

Using many small cores provide more thread level parallelism, but the impact of the serial scaling limits the achievable performance as the time taken to execute the serial section depends on the strength of the core. Hill and Marty [HM08] show that heterogeneous multicores can offer potential speedups that are much greater than homogeneous multicore chips. Heterogeneous cores that feature few very powerful cores, allow the use of an aggressive big core to speedup the serial section to amortize/reduce the impact of serial scaling on the overall performance. By looking at the relative benefits of the larger serial core (relative core strength) i,e $\frac{t_{ser\_little}}{t_{ser\_big}}$, we can infer whether the application has a potential to benefit by using a hybrid core. If the fraction is significantly greater than 1, then the serial part of the application executes faster in bigger core and we can expect some potential improvement using the hybrid.

In this thesis, we consider a heterogeneous core consisting of one big Xeon like core and many small Xeon-phi like cores (Asymmetric cores). As Xeon Phi's area details are still unavailable, we do a pessimistic area-performance analysis with the details of Out-of-Order Xeon (Big core) and In-Order Knights Ferry (Little core) as stated in [HBT13]. Die area per core comparison is around 1:3 between Xeon and Knights Ferry i,e 3 little cores can be built in the area of 1 big core. We show 3 different area-performance plots in Fig. 3.7 where x-axis is the area of Xeon, Xeon-phi and Hybrid equivalent of Xeon area and their respective performance in y-axis. The plots are 1. Xeon (all big Xeon cores), 2. Xeon-Phi (all Knights Ferry small cores), 3. Hybrid (One big Xeon core which executes serial section and rest Knights Ferry small cores). The total number of Xeon equivalent cores of Xeon-Phi and Hybrid cores are shown in Table 3.6. We will focus only on those benchmarks for which the experiments were carried out with the same input set size in both the platforms as mentioned in Sec.3.4.2.

From Table. 3.3, we can see swaption does not have any serial section and hence will not benefit from hybrid architecture. On contrary big Xeon cores has good speedup due to their well scaling parallel section. Fluidanimate has a very low core strength

Xeon (Big cores)	Xeon-Phi (Small cores)	Hybrid(Small + Big cores)
8	24	21 + 1
16	48	45 +1
32	96	93 + 1
64	192	189 + 1
128	384	381 + 1
256	768	765 + 1

Table 3.6: Total Xeon equivalent cores in Xeon-Phi and Hybrid cores

and will not have reasonable gains from hybrid core. Here, the little and hybrid cores perform better as the application scales better in Xeon-phi.

In canneal, the serial section is fixed and Xeon core is $3X$ faster than the Xeon-Phi. Therefore, by using a hybrid core we can get better speedup. Moreover, a good parallel section scaling ($PPS = -0.873$) with many little cores has better performance. Survey also gains better performance using a hybrid core as the big core executes the serial section $2X$ faster than the little core. But, the performance of Xeon is poor due to the poor parallel section scaling.

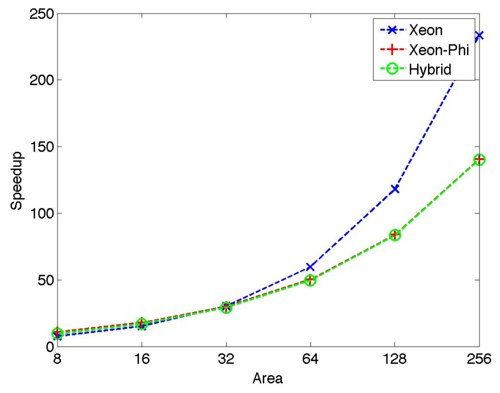
3.7 Summary

Future many-core designs will demand programs with very high degree of parallelism. The available parallelism might be restricted due to the programing techniques used in the application i,e application inherent behavior or due to the weak underlying hardware which cannot exploit the inherent parallelism in the application.

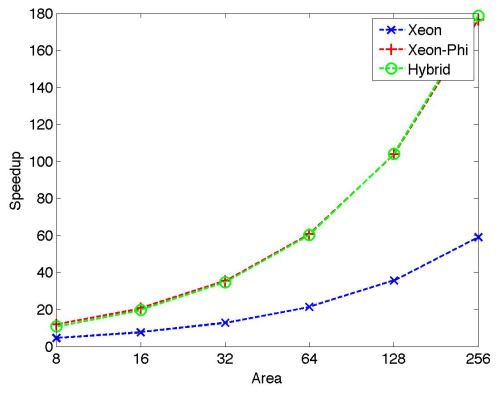
In this chapter, we developed a methodology to measure the serial and parallel execution time separately using *tiptop*, then applied non-linear regression on the experimental data to obtain the parameters for the SPS model. Our SPS model allows to capture various scaling behaviours of applications at an abstract level depending on the input size I and the number of cores P. We recall here this simple modelization:

$$t(I,P) = c_{seq} I^{as} P^{bs} + c_{par} I^{ap} P^{bp} \tag{3.8}$$

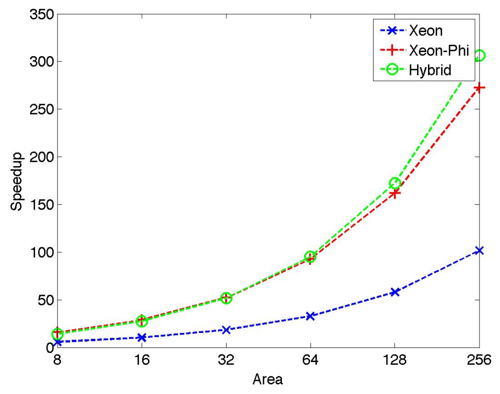
Through run-time monitoring, we have been able to empirically extract the parameters for parallel benchmark applications on a Xeon and Xeon-Phi many-core system. Our model indicates that there exist some applications that could scale well on a 1000 core processor (PPS close to -1) provided that the architecture is scaled in a balanced way. But, it also indicates that despite near optimal linear speed-up on the parallel part, global performance could be quite disappointing due to the scaling of the sequential part with the input set size.



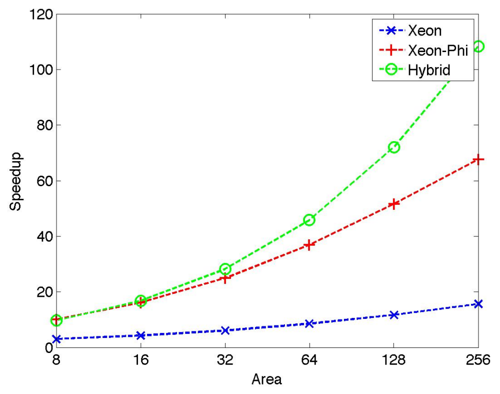
Swaption



Fluidanimate



Canneal



Survey

Figure 3.7: Area-Performance graph showing Hybrid architecture has better speedup with serial scaling.

From the application side, we hope that the SPS model will help application designers to understand how their application performance will scale with the number of cores and with the input set problem on future many-cores with few experimental runs providing the information about the scalable behavior of an application in a given architecture. On the architectural side, the SPS model can be used to study the architecture design space i.e whether big/small homogeneous cores or a mixed heterogeneous cores are needed for the application because providing a powerful core to the growing serial part will improve the overall performance of the program which is not discussed in this paper due to space limitation.

Chapter 4

Designing/Exploring core choice of future many-cores

In this chapter, we focus on understanding the difference between the inherent program behavior of the serial and parallel parts of a multi-threaded application. Then, utilizing a trace based simulation, we explore the design space of the core for many-cores processors (area-performance tradeoff) primarily focusing on the resource requirements of the serial and parallel cores.

Amdahl's law [Amd67] says that, we cannot go faster than the serial section of the program though we might have infinite processing resource. Therefore, to obtain optimal performance in many-core era, one must exploit all levels of parallelism i,e Thread Level Parallelism (TLP), Data Level Parallelism (DLP) and Instruction Level Parallelism (ILP) : TLP by running more concurrent threads, DLP by performing vector operations and ILP by making sequential core faster. In the previous chapter, we showed that the performance of serial and parallel fraction of an application depends on the underlying architecture. We also demonstrated how the applications in which the serial part becomes a bottleneck with larger input set size can benefit from asymmetric architecture.

Major emphasis in this chapter is to better understand the inherent program characteristics of the parallel and serial code sections in multi-threaded applications. Inherent program characteristics are used to get better insight on the respective resource requirements of the serial and the parallel parts of the application. If the requirements are similar, homogeneous multi-core processors should be used, where, sequential and parallel section will be executed in the core with similar resource. If not similar, heterogeneous multi-cores with complex core targeting the sequential section and simple cores targeting the parallel sections seems more adapted. This trend is already setting in the industry with the companies like Intel and ARM introducing Xeon-phi (many small cores) and ARM Big.Little (Few big cores and few small cores) respectively. However, the design space is huge as we move from single core to multi/many-core and is a difficult task to find an optimal design point. Therefore, our main focus is to explore the potential many-core parallel applications and find out whether they have some similar pattern or signature in the inherent behavior of serial and parallel section of an application which will help the architects in determining the proper use of the silicon area.

4.1 Motivation

The main motivation behind this contribution comes fom the hardware side in answering **what is the optimal micro-architecture of every core in many-core era?** We address this issue, by showing the difference between parallel and serial parts of a multi-threaded program and the kind of resource they demand.

To start with, we used multi-threaded pintool to analyze the dynamic instruction mix of the serial and parallel parts. We categorized the instructions into 5 categories such as *computation* - includes all the arithmetic and logical operations, *data-transfer* - includes all the memory bound operations, *branch* - accounts for all the control flow instructions, *stack* - accounts for push, pop operations and *semaphore* - accounts for the atomic operations. They are captured in terms of dynamic instruction mix of serial and parallel sections of 8 different applications and are presented in Figure. 4.1 and 4.2. Comparing both the figures, we infer that, the arithmetic instructions are considerably[1] more in parallel part while serial part has comparatively more data-transfer[2], branches and stack instructions. Another striking difference is the amount of vector instructions (SSE) in the serial and parallel parts which are shown in Figure 4.3. All parallel kernels have high vector utilization with *particle* being the exception which is due to the inherent nature of the program.

From theses characterizations, we deduce that, at the instruction level, the composition of parallel and serial sections are different. We further show that, the resource requirements are also different between these two parts in Section 4.4.

4.2 Related Work

Design space exploration of a processor at a very fine level results in an application specific cores where the processing elements are customized to specific applications by removing all the components that are not used or under-used by the application. Conservative cores [VSG+10] stands as a representative example for the application specific design space exploration by automatically synthesizing the cores from applica-

[1]Average arithmetic operations in serial section = 36%, parallel section = 50%. We can see that 5 out of the 8 benchmarks have more than 50% of arithmetic operations in parallel section.

[2]Average data transfer operations in serial section = 37%, parallel section = 33%. This 4% might look marginal but with the memory access latency into consideration, it can have a significant impact in the run-time depending on the memory hierarchy.

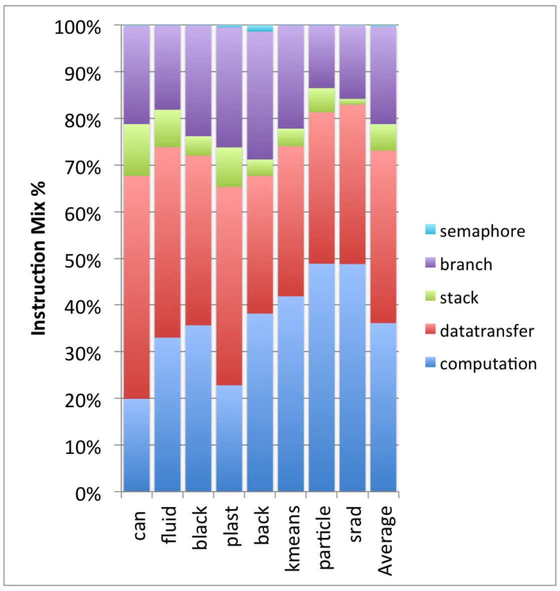


Figure 4.1: Instruction mix of serial section.

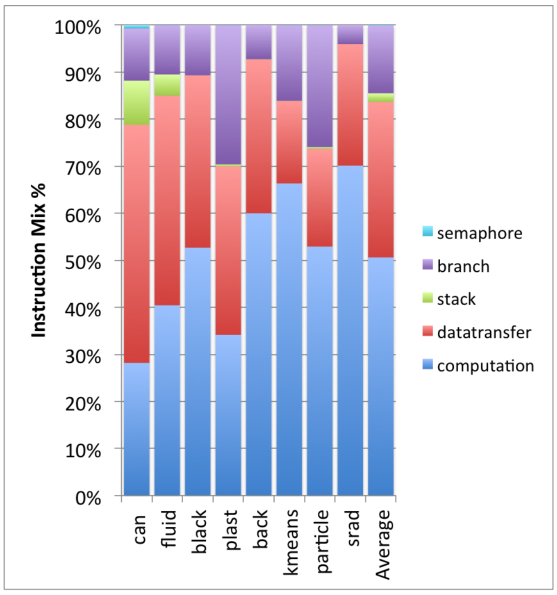


Figure 4.2: Instruction mix of parallel section.

tion source code. These application specific processor cores mainly target improving the energy-efficiency by increasing the parallelism. As a result, per-core computation power requirement reduces and more computation is possible under the same power budget. On a coarser level, there are processors which target a specific set of applications. For example, GPUs target group of applications with high arithmetic intensity.

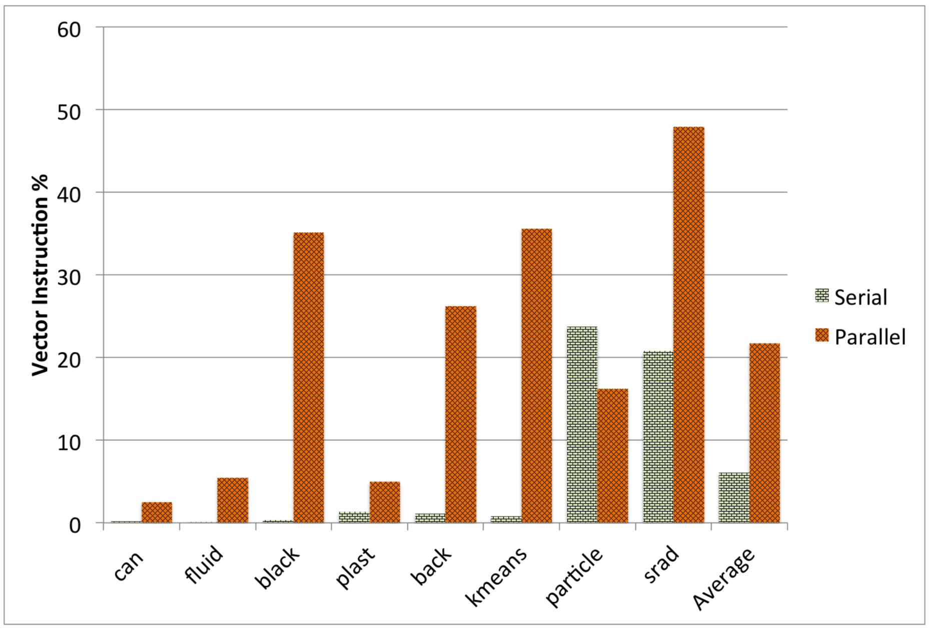


Figure 4.3: Percentage of vector instructions in serial and parallel part.

One of the earliest design space exploration work was by Farrens *et al* [FTP94] who studied the area efficiency of processor by comparing a single core with a large cache to a multi-core architecture with smaller private cache and concluded that multi-core with smaller cache have a good area-performance trade-off. Similar study was conducted by Huh *et al* [HBK01] on efficient design of Chip Multi Processor (CMP), where, they studied the area-performance trade off of CMP implementation to find how many cores future CMP should have and what kind of core should they be considering different factors like processor organization, cache hierarchy, off-chip bandwidth and application characteristics. They concluded that, by fixing the cache size and the number of cores *a priori* might result in poor performance across different application classes. Hence, the future CMPs should be adaptable, where, the processors will have mechanisms to adapt to the applications available parallelism and their resource needs.

Along with homogeneous CMPs, the research community also proposed some exploration results for heterogeneous CMPs. The initial exploration on the heterogeneous CMPs were carried out with existing architectures, either with different generation of same processor family or by changing voltage and frequency of the processor ([BRUL05], [AGS05]). Later Kumar *et al* [KTJ06], studied the core architecture optimization for heterogeneous CMPs. They arrived at multiple conclusions like the best performing general purpose cores may not be suitable for heterogeneous CMP, instead, each core should be tuned for a class of applications with common characteristics when executing a multi-process workload.

In this thesis, we also focus on obtaining optimal area-performance trade-off like the previous works, but, our design space exploration goals are bit different. Instead of looking at individual applications requirement and clustering them w.r.t to their algorithmic similarity (example: stencil, streaming etc), we intend to explore the inherent characteristics of serial and parallel sections of a group of multi-threaded applications and understand their hardware requirements respectively. Closest to our study in this chapter was by Tullsen *et al* [ANM+12],where, they show that CPU and GPU sections of a program has different characteristics and the design of CPU should be redefined for better CPU-GPU integration. They analyzed the non-GPU code characteristics and proposed modification required in CPU design based on sensitivity of the window size, branch predictors, vector units and prefetchers used.

4.3 Experimental Methodology

The methodology for design space exploration should start with a set of workloads and a set of constraints on the processor. The design space of even a single core is large with a possibility to experiment by changing various architectural parameters. Moving towards multi/many core the exploration becomes more complex because the design space extends to multiple cores and on top of it, the best achievable performance depends on software/hardware co-design of the application. From the motivation in Section 4.1, we inferred that the program composition of the serial and parallel sections are different. Hence, we make some assumptions to make the problem simpler by exploring the resource requirement like the number of computation units, type of branch predictor or size of the cache required by the serial and parallel section in an isolated way. The methodology adapted by us may be bit naive for a commercial design, but even in that case we believe this methodology would find a design closer to the best achievable design.

4.3.1 Benchmarks

In this study, we focus on applications that will be executed on future many-cores. Therefore, we chose benchmarks which are parallelized using shared memory model. We selected applications featuring parallel and serial section where both the sections do meaningful work. We consider 8 applications with the criteria that the total number of instructions executed in the serial section is considerably large i.e, the serial part should execute 1 billion instructions at the least and contributes to 10% of the total run-time instructions at the minimum on the chosen input set.

After evaluating the data parallel and balanced programs from PARSEC, RODINIA and PLAST benchmark suites, we chose 3 applications from PARSEC benchmark suite[BKSL08]: *Blackscholes(black), Canneal (can), Fluidanimate (fluid)*, 4 applications from Rodinia [CBM+09] benchmark suite: *Backprop (back), Kmeans (kmeans), Particle Filter (particle), SRAD (srad)* and PLAST bioinformatic benchmark [NL09].

We also verified from the source code that, these programs do meaningful work in the serial part just more than reading input file and allocating memory. The serial activity in these benchmarks are further explained in detail:

1. **Blackscholes:** The Blackscholes equation is a Partial Differential Equation that describes, how, with a given set of assumptions, the value of an option changes as the price of the underlying asset changes. The serial part in this benchmark involves the initialization phase where the data are calculated, segregated and stored in private arrays to be used by the parallel part for further computation.

2. **Canneal:** This program simulates cache-aware annealing to optimize routing cost of a chip design. The serial section does a big job in creating the netlist from the input file which will be used by the parallel kernel and the time taken by the serial section to build the netlist increases with higher input set size.

3. **Fluidanimate:** This is a physics simulation which animates the fluid flow by solving the Navier-Stokes equations using Smoothed Particle Hydrodynamics (SPH). In the program, the initialization phase is compute bound with lot of variables and coefficients being computed for the further use in parallel section. We also noticed serial section in terms of visualization barriers in between parallel fluid simulation sections which can be a potential scaling bottleneck.

4. **Backprop:** Back Propagation is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network. In this application the serial part consists of net list creation where new fully-connected network is built from scratch, with the given numbers of input, hidden and output units. Also the error computation part in between the successive execution of the parallel kernals contributes to the serial section.

5. **Kmeans:** K-means is a clustering algorithm used extensively in data-mining that identifies related points by associating each data point with its nearest cluster, computing new cluster centroids, and iterating until convergence. In this application, the serial part consists of memory allocation, splitting up and arranging data into arrays, performing array reduction, replace old cluster with new cluster etc which are computationally intensive task of the application.

6. **Particle Filter:** The particle filter is a statistical estimator of the location of a target object, given, noisy measurements of that target's location and an idea of the object's path in a Bayesian framework. This is used for video surveillance in the form of tracking vehicles, also in biological applications like tracking cells. In the program, the video sequence is generated with noise from the given data which contributes to the serial section and is computationally intensive.

7. **Srad:** It is used as a diffusion method for ultrasonic and radar imaging applications based on partial differential equations (PDEs). SRAD consists of several pieces of work like image extraction, continuous computation and image compression. Out of these functonalities resizing image, extraction, compression and

saving of image are implemented serially which contains equal computation and memory operations.

8. **PLAST:** Parallel Local Alignment Search Tool (PLAST) for Database Comparison is a bio-informatics algorithm to detect similarities between two protien sequences. PLAST implements a 3 step, seed based algorithm which involves indexing, ungapped extension and gapped extension. The indexing part of this algorithm is exclusively implemented serially and grows when bigger workloads are executed.

4.3.2 Methodology

We used the multi-threaded PINTOOL [LCM$^+$05] to analyze our applications with dynamic instrumentation. We analyze the Micro-Architecture Independent [HE07] characteristics of serial and parallel sections of these application and then imply the micro-architectural constraints to identify the resource requirement of serial and parallel parts individually. Therefore, every thread of the application is analyzed independently and results are obtained per thread-wise. Our main goal is to find out the difference in the code characteristics and resource requirement as mentioned before in Section 2 between serial and parallel sections of a multi-threaded program. All our programs were compiled with GCC 4.7.2 and -O3 optimization. We used large input set size available from the benchmark suites and the results are reported from complete program execution. The details of the input set used are provided in Table 4.1.

Benchmarks	Input Size
Blackscholes	65,536 options
Canneal	400000 elements, 100000 swaps per step, 512 steps
Fluidanimate	500,000 particles, 80 frames
Backprop	1048576 nodes
Kmeans	819200 data points, 34 features
Particle filter	-x 128 -y 128 -z 200 -np 1000
SRAD	1024 x 1024 datapoints
PLAST	-i coina.fa -d query.fa

Table 4.1: Input set parameters used for different applications under study.

We ensured that the chosen benchmarks are load balanced and execute equivalent work in all the parallel threads. Among the chosen 8 applications, 4 are parallelized using Pthread library (black, can, fluid and plast) and rest 4 (back, kmeans, particle, srad) with OpenMP library. In the pthread based programs, it is easy to differentiate parallel and serial section. The serial section or the main thread of the program always executes as the first thread, which in turn spawns the requested number of threads in runtime (in total N+1 threads). As we analyze the activity threadwise, the muilt-threaded pintool reports the analysis on the program on per thread basis, where, first thread corresponds to serial section and the others are parallel. On the other hand, OpenMP spawns only N threads (main thread does its work and also executes one parallel thread), therefore, we extract the parallel contribution from the main thread to know the serial activity[3]. Hence, no source code modification or marking in the code was required.

4.4 Difference between serial and parallel section

We study the difference between the serial and parallel sections by analyzing the micro-architecture independent characteristics [HE07] of each section. Micro-architecture independent characteristics infers the inherent behavior of the program, using which, we can explore the hardware requirement for serial and parallel sections. We consider the following characteristics in our study:

1. potential Instruction Level Parallelism (pILP) available in serial and parallel sections to know the minimum number of independent instructions that can be executed in parallel in each part of the program. Inturn, pILP gives an idea about the minimum number of execution units required by the application.

2. Always taken control instructions in serial and parallel parts to know the taken rate or transition rate of the branches. Branch Mis-prediction per Kilo Instructions (MPKI) of serial and parallel sections infers whether the application needs simple or complex branch predictor.

3. Memory foot print or working set size of serial and parallel sections to know number of different memory blocks touched by the application. This helps in understanding the cache requirement of each section.

Performance of a multi-threaded program in many-core era depends not only on Thread Level Parallelism (TLP) available in entire application but also on Data Level Parallelism (DLP) and Instruction Level Parallelism (ILP) available per thread. Limits of ILP has been studied for long now with first work from Wall *et al*[Wal91] considering the limitations imposed by register renaming, branch prediction and window size. They concluded that available ILP lies in the range of 5-7. In this section, we study ILP limits in parallel and serial sections to explore what will be the optimal micro-architecture in the many-core era. Also, the size of the cache and the nature of the cache (private or

[3]We would like to clarify that for our analysis this approximation is acceptable because we work on one IPC model so the absolute subtraction does not distort the result

shared) in the memory hierarchy determines performance of the thread by improving data reuse and by hiding memory latency.

4.4.1 ILP

Instruction-level parallelism (ILP) is a measure of how many operations in a computer program can be executed in parallel if we have infinite resources. In reality, we are confined by area and power constrains which bounds us with limited resources. The most significant hardware parameters in determining the available ILP in a program are 1. Window-size and 2. Issue Width.

1. *Window size (WS)* - The window size bounds how far ahead the processor can search for ILP. For example, if the $Windowsize = 32$, then we can look for ILP available in the set of successive 32 instructions in the instruction stream. In the recent Intel Haswell processor, WS=192 [HKO+14].

2. *Issue-width (IW)* - The maximum number of instructions that can be issued (i.e., commence execution) during the same cycle. The issue-width bounds the maximum available Instruction Per Cycle (IPC). In the recent Intel Haswell processor, IW=8 [HKO+14] as it has eight heterogeneous execution ports.

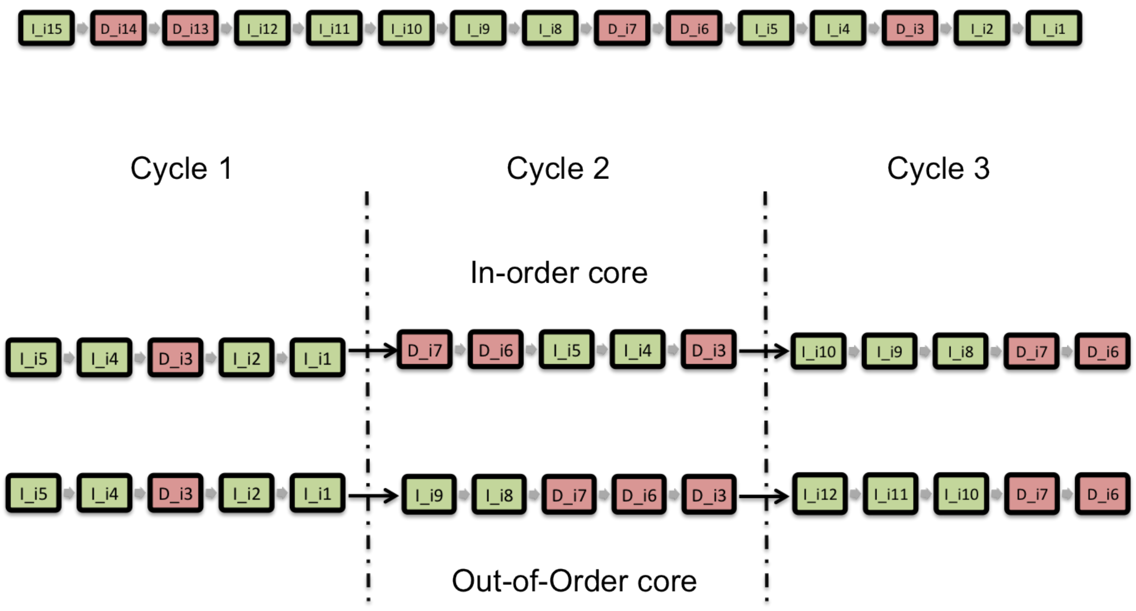


Figure 4.4: Shows general philosophy of IO and OOO execution of the instruction stream with WS=5. Here I_ix denotes an independent instruction, D_ix denotes a dependent instruction.

potential ILP (pILP): is defined as the maximum number of instructions that can be executed in parallel for a considered window size. The amount of pILP varies widely depending on the type of code being executed.

Our aim in this section is to understand the pILP in the serial and parallel sections of different applications when using In-Order (IO) and Out of Order (OOO) core model respectively. To measure the pILP in shared-memory multi-threaded applications, we begin by assuming an ideal processor core model with a unit latency instructions which has perfect cache, perfect branch predictor and infinite physical registers.

4.4.1.1 IO pILP

In-Order model assumes unlimited instruction window size, functional units to analyze the micro-architecture independent characteristics of the application. The philosophy of the model is shown in Figure 4.4 and employs data dependency analysis. We can observe that, in the in-order core, only the independent instructions from the head of the window till the first dependent instruction in the window gets executed in every cycle and the dependent instructions will be executed in the next cycle. For example: In the In-Order core part of the illustrated Figure 4.4, at $cycle1$, I_i1 and I_i2 are executed as they are independent instructions at the window head before the first dependent instruction D_i3 and at $cycle2$ D_i3 becomes independent, therefore, D_i3, I_i4, I_i5 are executed from the head of the window till the first dependent instruction D_i6. This model gives the lower bound of issue width (corresponds to minimum number of execution units) required by the serial and parallel parts respectively. Our experiments shows that in all applications serial code has more successive independent instructions than the parallel code as demonstrated in Figure. 4.5. Among the considered applications, average serial $pILP = 3.3$ and average parallel $pILP = 2.3$. This suggests that, the core executing serial part will benefit with more execution units than parallel part.

4.4.1.2 OOO pILP

In Out-of-Order model, we assume a fixed window size and issue-width to obtain pILP. The philosophy of the model is shown in Figure. 4.4 with $WS = 5$, where, all the independent instructions in the window are executed in every cycle. For example: In the Out-of-Order core part of the illustrated Figure 4.4, at $cycle1$, I_i1, I_i2, I_i4, I_i5 are independent instructions which can execute if there are sufficient executable units and at $cycle2$, D_i3 (becomes independent), I_i8 and I_i9 are executed and hence forth. This is further limited by considering the issue width (number of execution unit) available. In our analysis, we assume different window sizes of: 32, 64, 128 and 256 in-flight instructions and issue width=8. Extractable pILP for small (IW=8,WS=32) and large (IW=8,WS=256) configurations of OOO cores are shown in Figure. 4.6. We can observe the following:

1. Bigger the WS, more the pILP but still not sufficient enough to utilize IW=8.

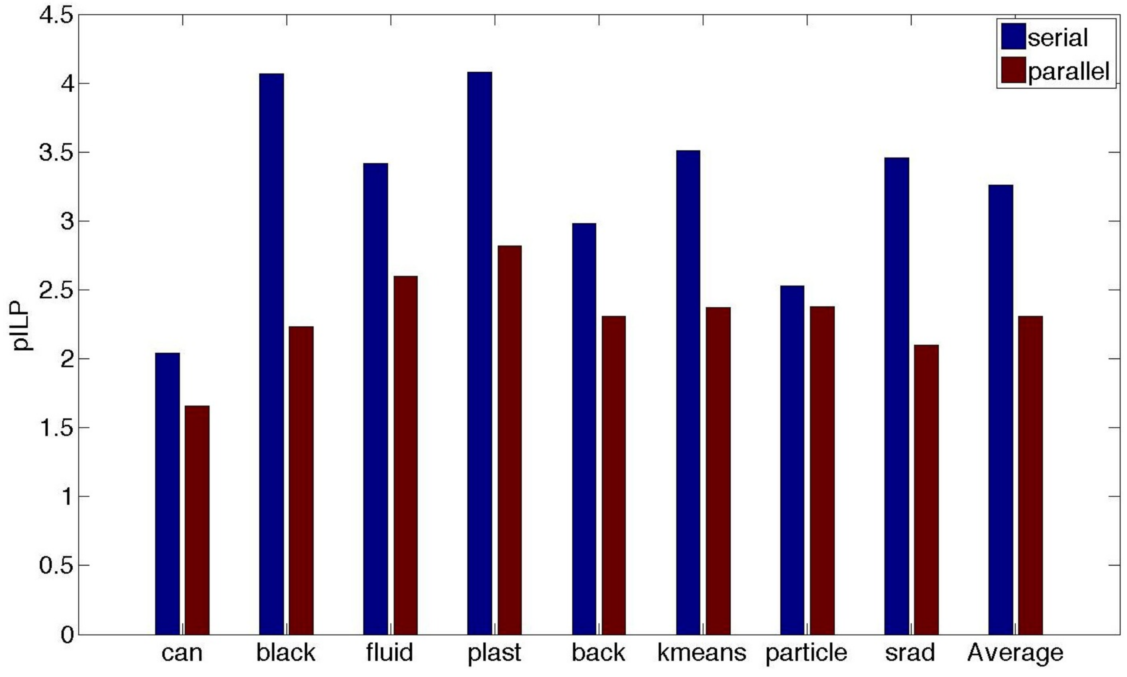


Figure 4.5: IO pILP of serial and parallel section.

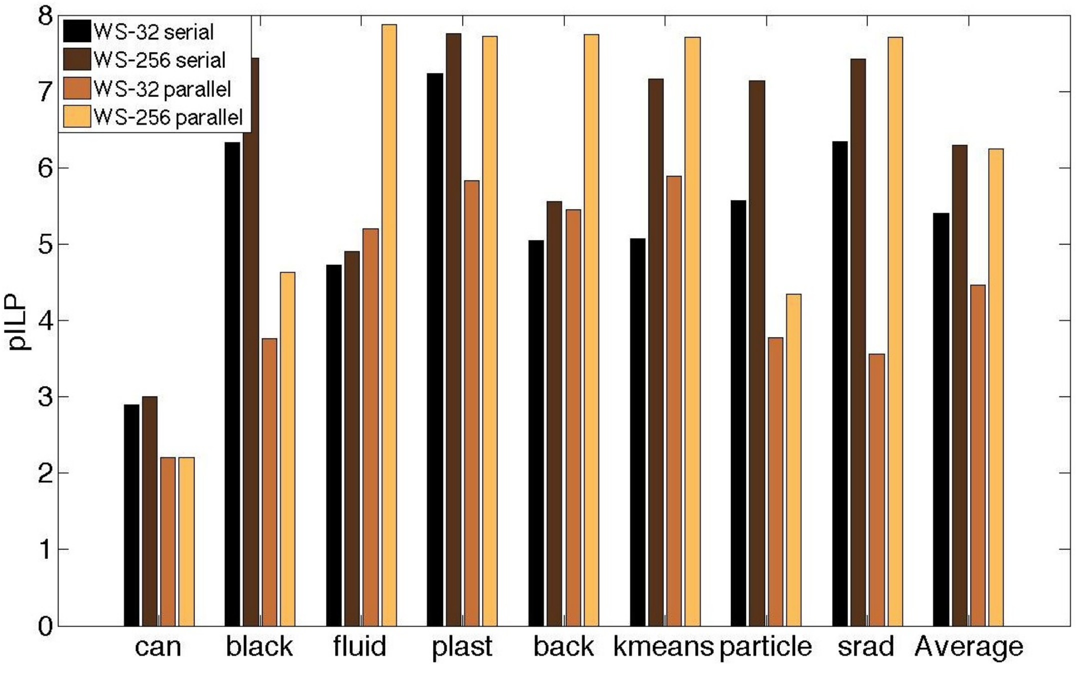


Figure 4.6: OOO pILP of serial and parallel section.

2. Though serial section has higher pILP than parallel section, the difference between them reduces as we increase the WS.

4.4.1.3 Inference

From pILP study, we infer that, the extractable pILP is different for OOO and in-order cores. Figure 4.7 shows the trend of how $pILP$ grows in the serial and parallel part as we move from IO core to OOO with $WS = 32/64/128/256$. In general, by looking at the average pILP of both IO and OOO cores, we see that the parallel part uses functional units at a faster rate than the serial part as the window size grows, i.e, more independent instructions can be executed in OOO core with larger WS. But, the performance/area/power cost and complexity of the architectural technique to exploit pILP also grows exponentially for OOO cores. In Chapter 3, we demonstrated how an asymmetric heterogeneous cores with larger serial core and many smaller parallel cores improved the overall performance of an application. Applying the similar train of thoughts and from the pILP experimental results, we postulate that in the future many-core serial part will need a wider issue core and many small IO cores or OOO cores with narrow issue width and small window size will be more appropriate for the parallel part.

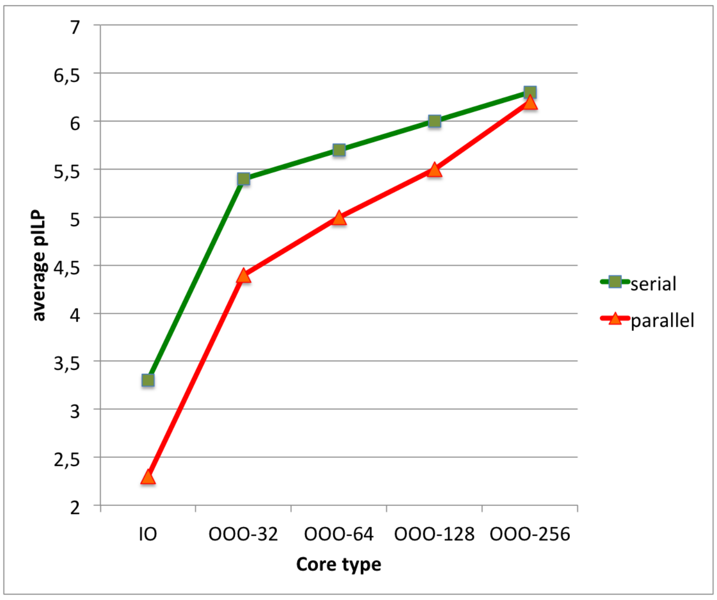


Figure 4.7: Shows the average pILP trend of the serial and parallel sections of the applications under study when we move from IO to OOO core.

4.4.2 Branch predictor

Control instructions also limits the available ILP in an application [LW92]. With deeper pipeline and the potential penalty of branches increasing, we need dynamic branch prediction techniques with high accuracy to speculatively execute the instructions. As we aim at exploiting ILP in the cores executing serial and parallel sections, the prediction capability of the branch predictor becomes very important.

For instance, a study from 2009 [RWJ09] finds that, for the Intel Xeon E5440, a perfect branch predictor would yield a performance improvement of 26.0%, while halving the average number of mispredictions per 1000 instructions (MPKI) from 6.50 to 3.25 would improve overall performance by 13.0% . For future generation processors with large instruction windows and large issue width, the effect of branch prediction on performance will be even more pronounced.

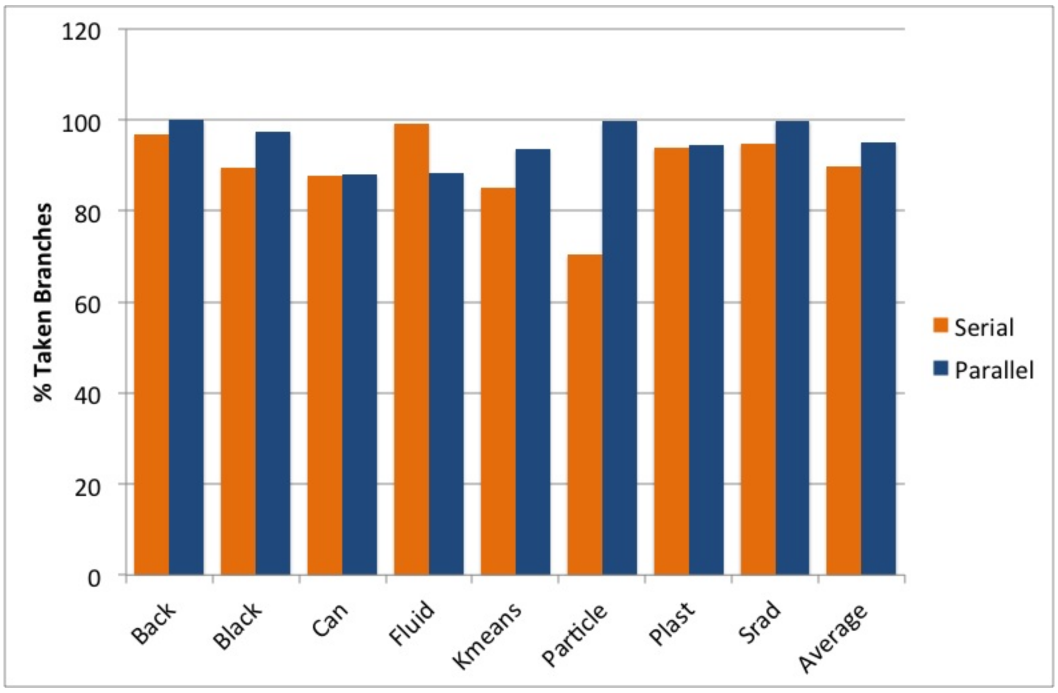


Figure 4.8: Percentage of taken branches in serial and parallel part.

To study the Micro-architecture independent characteristics of conditional branches it is necessary to know how often the branches are always taken. For the applications under study, we performed an analysis using the multi-threaded PINTOOL to find the percentage of taken branches in the serial and the parallel parts. We find that on an average the parallel part has 7% more taken branches than the serial part in the considered applications. Figure 4.8 shows the inherent branch behavior of the applications in serial and parallel parts respectively. We can very well notice that except for $fluid$ rest all applications have more taken branches in the parallel part which can be attributed to the inherent nature of the program. Continuing this, we explored different hardware branch predictors based on their complexity to study the achievable prediction accuracy for the serial and parallel parts respectively.

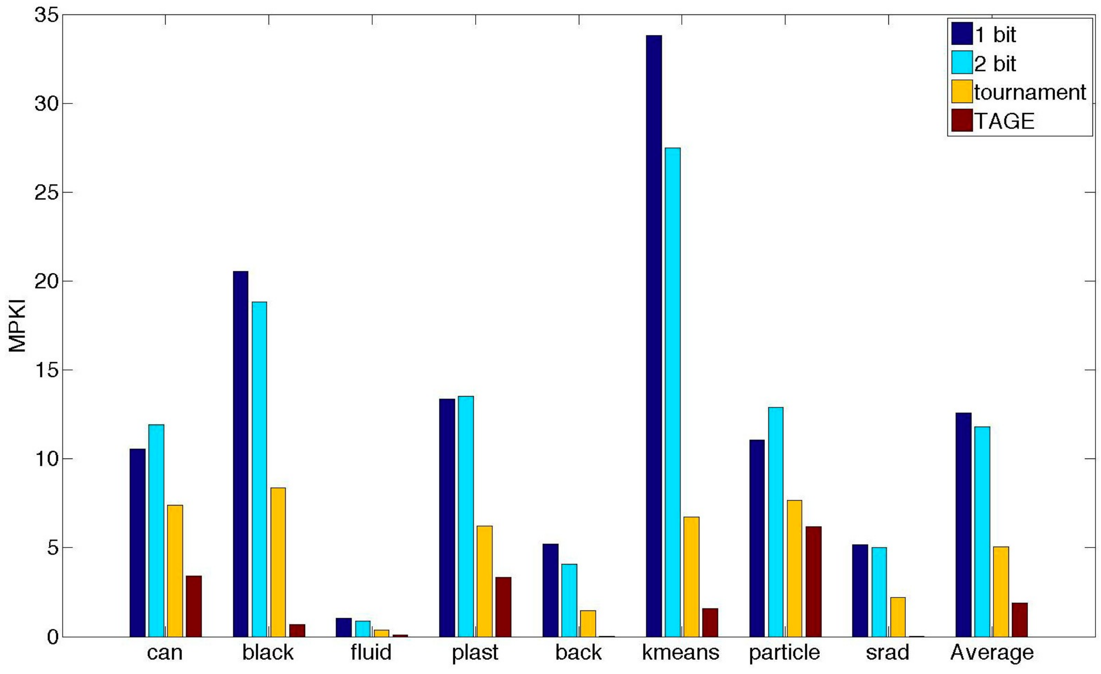


Figure 4.9: Branch behavior of serial part on different predictor with 8KB storage size.

In our study, we have compared the MPKI achieved on serial and parallel sections using simple (1-bit [Smi81], 2-bit [YP91], tournament [McF93]) and complex (TAGE [Sez11]) branch predictors. TAGE is considered as the best performing branch predictor proposed during the last decade. TAGE handles both conditional and unconditional branches. Experiments were performed with both simple and complex predictors with storage capacity of 8KB among all predictors. Therefore, only difference comes from the complexity of implementing the logic. 1-bit will be easier to implement than for tournament.

4.4.2.1 Inference

From Figure. 4.9 & 4.10, we see that, the TAGE predicts well in parallel section of most applications with MPKI ≤ 1 but the serial section still has some difficult to predict branches in *can, plast, kmeans and particle.* Tournament predictor which is less complex to implement than TAGE also works well in most parallel applications except *fluid*($fluid$ was an exception among the benchmarks with more taken branches in the serial section than parallel) with MPKI ≤ 4. Therefore, a fair choice would be to use a simple branch predictor which are simple to implement with good accuracy for parallel part and a complex predictor like TAGE for serial part. In Figure 4.10 we can see that going from 1-bit to tournament we gain 2 MPKI which should be around 8-10% with respect to the reverse engineering results from [RWJ09].

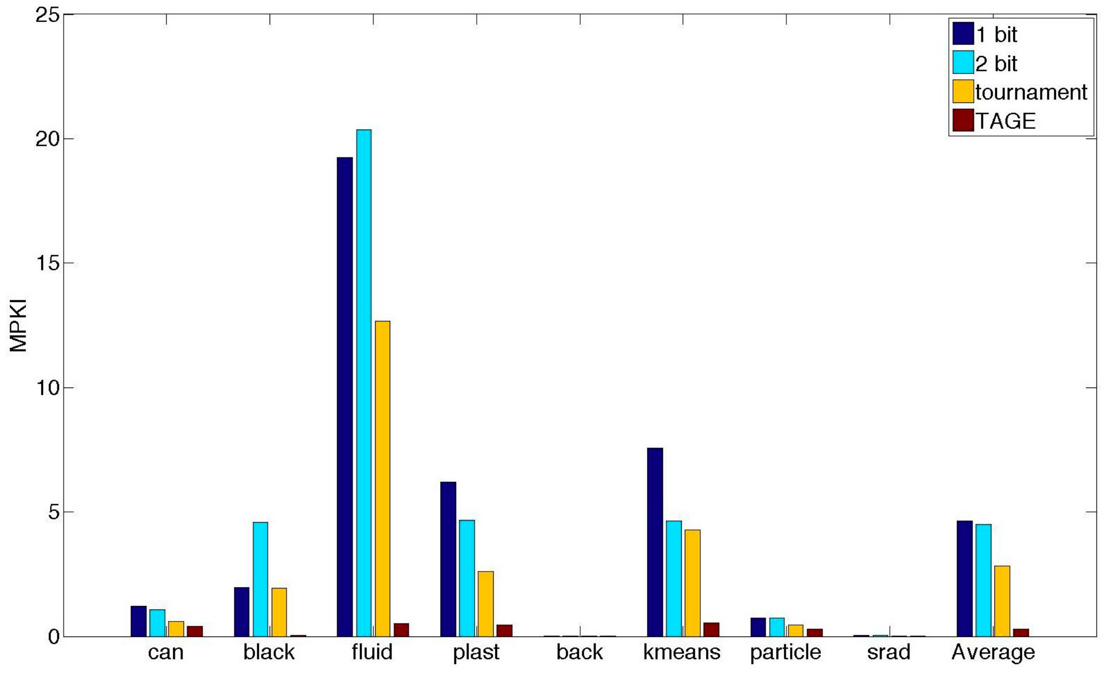


Figure 4.10: Branch behavior of parallel part on different predictor with 8KB storage size.

4.4.3 Data cache

Private caches have become part of the CPU in multi-cores. Cache is used by processor to reduce the average time to access data from the main memory by storing the frequently accessed data nearer to the core. By understanding the temporal locality behavior of a programs' data, we can determine the cache requirement of serial and parallel parts of the program. We use a multi-threaded PINTOOL based cache simulator for this purpose which simulates a 8 way associative cache with round robin replacement policy with 64 byte blocks. We studied the MPKI[4] with cache size varying from 32KB to 32 MB to understand the cache requirement of the serial and parallel parts respectively.

4.4.3.1 Inference

From Figure. 4.11, we observe that the cache requirement for serial and parallel section are different. Most of the considered applications do not require large private cache for

[4]Misses per Kilo instructions

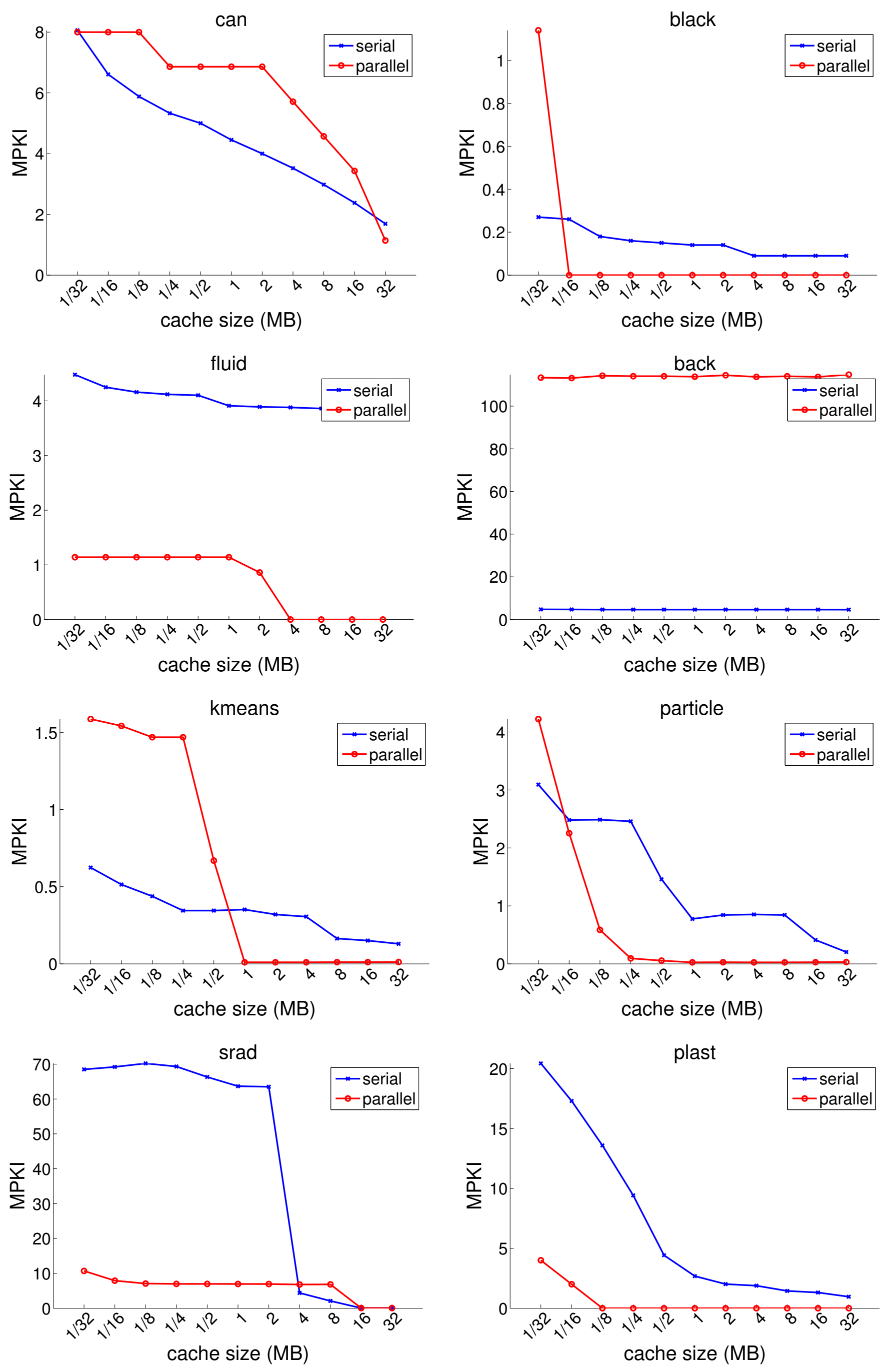


Figure 4.11: The figure shows the cache performance of each workload as a function of cache size. The y-axis presents workload MPKI and the x-axis presents the cache size in Mega Bytes (MB). All workloads were run to completion using the reference input sets.

the parallel section (more than 1 MB) except *can, back and srad*. Moreover, parallel section of *back* does not even benefit from large cache as the MPKI remains same with larger cache size. On the other hand, most serial section needs large cache and benefit from the growing cache size. Therefore, the cores executing the serial section should have larger caches than the ones executing parallel section in general. *Fluid* (serial) and *back* (parallel) does not even benefit from 32MB because either their data are not reused or they are so big that it does not fit in this cache size.

4.4.4 Future many-cores

From the micro-architecture independent study, we can see that the inherent behavior of serial and parallel parts are different and have different resource requirements with respect to different component of the core like number of execution units, branch predictors and private cache size etc. Though in specific these parameters are application dependent, we found a common trend to make some fair decisions on the behavior and resource requirement for these sections.

- *Serial section* has 1. high average pILP both in In-order and OOO model, 2. more conditional branch instructions which are difficult to predict and 3. large working set size. Therefore, core executing the serial section can be a wide issue OOO core with complex branch predictor and large sized private cache.

- *parallel section* has 1. low pILP in In-order and OOO model along with high chance of exploiting TLP with many small cores, 2. easily predictable branches and 3. smaller working set size. Therefore, core executing the parallel section can be a small issue OOO core or IO core with simple branch predictor and medium/small sized private cache.

Thus, the future many-core should be heterogeneous with many small cores executing the parallel section and few complex cores executing the serial section. With this study, we affirm the conclusions derived from analytical models ([HM08], [NBS14]) on the usage of many small cores and few high performance big cores in many-core era.

4.5 Summary

With the hardware and software requirements unclear for the many-core era, we have taken a fair step in this chapter in determining the hardware requirements by analyzing the inherent characteristics in the serial and parallel part of the MT application. Though the real world is much more complex, we have attempted to answer the design space exploration problem of many-core at high level and analyzed how the core and uncore characteristics can guide the architect in designing a many-core targeting a class of applications.

The contribution in this chapter can be classified into two sub-problems. 1. To understand, whether the serial and parallel sections of a multi-threaded application are similar or different? From our characterization study, we found that they are different. Serial parts are memory intensive with more hard to predict control instructions and the parallel parts are usually computationally intensive. 2. What kind of core should be used for the serial and parallel parts? From our high level design space exploration study, we found that serial part needs wider issue core with complex branch predictor and large private cache and in contrary, the parallel part might perform fairly with a narrow issue core with simple branch predictor and comparatively smaller private cache. Our overall study suggests that serial and parallel sections are different: they might require different processors. Hence, we conclude that heterogeneous many-cores cores with few large cores and many small cores will be the optimal many-core architecture in the future.

Chapter 5

Conclusion

Diminishing performance returns, high power consumption and heat dissipation along with complex design led to the end of single-core scaling. With transistor size shrinking, the processor industry started integrating more transistors in a given area. In the recent trend, we can observe that these transistors are used to build more cores in a single chip resulting in multi-core processors. Multi-core has become so ubiquitous that today we find them not only in the desktops but also in most of the small portable devices like mobile phones, tablets and even in cameras. Similarly, the high performance computing industry has moved from multi-core to many-core and the last few generations of supercomputers like Stampede, Tianhe-2A host many-core processors along with the traditional multi-cores as accelerators.

Many-core era poses multiple design challenges both from software and hardware perspective. From the software side, the main challenge is to build parallel applications using scalable programming models that are portable across multiple platforms and can exploit the computing power available in the many-core. From the hardware side, the primary challenge is to build an area and power efficient cores which are generic and can execute the application without much of fine tuning and platform related optimizations. Additionally, there are secondary challenges like designing high bandwidth and low latency on-chip network, scalable memory consistency models etc that completes the many-core system. We discussed these design challenges in detail in Chapter 1 and postulated that when the performance is evaluated over the entire application and not just the parallel kernel a very basic scalability bottleneck may occur due to the serial section in an application.

Before building a new system, it is very essential to model it to study the achievable performance of the system. In the field of computer architecture, modeling a simulator to study a complete system is a time consuming process but can be accurate. In contrary, analytical models can be used to study the system quickly at a high level but are not very accurate. In chapter 2, we focused more on the performance evaluation techniques for the many-core system and discuss the state of the art analytical models that are used for performance prediction. Existing analytical models used to predict parallel application performance either considers serial section is constant or ignores the impact of serial section.

In chapter 3, we propose our Serial/Parallel scaling empirical model that captures effect of serial section in determining the application scalability and achievable performance. Our empirical model shows that first of all, serial section is not negligible in all parallel applications. Additionally, explicit serial section adds up to the performance degradation of the parallel section resulting in lower performance than expected. We validate our model in 2 different architectures 1. many small core Xeon-Phi many-core processors with 240 logical cores and 2. few large core Xeon Westmere based many-core processors with 24 logical cores. We finally demonstrate how the applications with scaling serial section benefit if a heterogeneous many-core is built with few large cores and many small cores.

Another interesting issue that has always been a hot topic of research in computer architecture is the design space exploration of the processors. On a very higher level, this can be seen as exploration of the kind of architecture that should be considered to get a better area-performance-power trade-off for a group of applications. In Chapter 4, we explore the design space for the apt hardware for multi-threaded applications in the many-core era. We analyze the core requirement for the serial and parallel section separately to conclude that the inherent characteristics of these 2 sections are different and hence might need different core configuration in the many-core era.

In this thesis, we have mainly focused in answering the following 3 questions

- **Q1. In a multi-threaded program, is the serial part negligible or constant as assumed by the previous analytical models that are very widely used?**
- **Q2. How to model the application scalability of many-core systems?**
- **Q3. What is the difference between serial and parallel parts of a multi-threaded program? Do they require similar micro-architecture?**

Contribution

As a first contribution, we developed a methodology to study the execution time of the serial and parallel section individually in a multi-threaded application. We used *tiptop* [Roh11], a performance monitoring tool that can sample and log the periodic samples (no.of active cycles and instructions executed) from threads running in individual cores. The biggest advantage of using *tiptop* were 1. low sampling overhead, easily scalable to many-core monitoring and 2. it does not require any source code marking or instrumentation. Using the measured data, we built our Serial/Parallel Scaling (SPS) model for each application executed on the given architecture. SPS model uses both input set size and the number of processors used as a function in determining the execution time model of a multi-threaded program. Our empirical model is shown in Equation 5.1. The SPS model only uses 6 parameters to represent the execution time of a parallel application, taking into account its input set and the number of processors. c_{seq}, as and bs are used to model the serial execution time and c_{par}, ap and bp are used to model the parallel execution time. We compared the maximum achievable speed up of the multi-threaded applications with the existing analytical models proposed by Amdahl [Amd67], Gustafson [Gus88] and Juurlink *et al* (GSSE) [JM12] to show that in reality serial section has a major impact in application scalability of certain applications where the serial section grows with the input set size. There are more inferences from the model such as the percentage of serial and parallel part in a multi-threaded application is architecture dependent and hence the scaling factor of the fraction of serial and parallel parts cannot be determined without actually executing the application and measuring them. For serial section scaling applications using a homogeneous many-core may not be a good choice and hence we demonstrated the advantage of using a heterogeneous many-core using few large cores and many small cores to improve the over-all performance.

$$t(I, P) = c_{seq} I^{as} P^{bs} + c_{par} I^{ap} P^{bp} \tag{5.1}$$

As a second contribution, We analyzed the program inherent characteristics of the serial and parallel parts of the multi-threaded applications and found that they both have different signatures. Serial part has more memory intensive and control instructions, while parallel part has more vector and compute instructions. Then, we conducted a design space exploration to find out the resource requirement of the core executing serial and parallel parts of multi-threaded application using multi-threaded PINTOOL based trace simulators. We explored three main resource requirements of the core executing serial and parallel sections individually 1. number of execution units required, 2. type of branch predictor required and 3. size of the private cache required. With the experiments, we infer the following 1. Serial part has higher inherent potential Instruction Level Parallelism than the parallel part and, hence, will benefit from more execution units i,e using a wider issue core. 2. Serial part has more hard to predict branches compared to the parallel part and requires complex branch predictors to achieve high prediction accuracy. 3. Serial part has a bigger memory footprint than the parallel part i,e the re-usage of data read from memory is less frequent and, hence, demands a bigger private cache than the parallel part. Thus, many-core with many small cores to execute the parallel sections and few large cores to execute the serial section seems to be more appropriate and emphasizes that the future of many-core should be heterogeneous cores.

Recommendations for future work

In our first contribution, the prediction was mainly based on the empirical model (black box model) built on training set obtained from measuring the execution time of the serial and parallel section of a multi-threaded program. This model has a limitation, as, it partially captures the performance degradation due to the hardware bottleneck due to resource contention. In order to capture the other hardware bottlenecks similar modeling can be done with the training sets obtained from computation, memory access and core to core communication as shown in Equation 5.2. In current day processors, these information can be obtained from the Performance Monitoring Units available in the hardware as the number of events. By fitting the model individually for computation, memory access and communication we can construct a mechanistic model that can be useful in understanding the application scalability limiting factor when building a many-core system.

$$t(I,P) = t_{comp}(I,P) + t_{mem}(I,P) + t_{comm}(I,P) \tag{5.2}$$

In our second contribution, we explored the design space of the core with respect to the resources required by the serial and parallel sections of the multi-threaded program. We investigated 3 out of the 4 main requirements of the core like number of execution units required, type of branch predictors required and the size of private cache required. This can be extended with a study on prefetchers in serial and parallel cores to make the core design space exploration complete. We can obtain interesting directions by finding out the prefetching capability of the serial and parallel parts of the program which requires a detailed analysis [LKV12] on the strides of data accessed by these sections respectively. Further different hardware and software prefetching strategies ([LM98], [WBM$^+$03],[MFZY14]) can be studied for the serial and parallel parts that can enhance the performance of the serial or parallel core. I have done some preliminary experiments with prefetchers but due to lack ot time it was difficult to conclude on the results whether prefetchers will benefit serial and parallel sections. Prefetcher analysis needs lot of experiments and a lot more analysis to conclude something concrete because getting a good prefetcher metric like accuracy, coverage and timeliness does not really guarantee an improvement in the overall performance which will become an exhaustive study by itself.

In this thesis, we focused on performance evaluation completely based on area-performance trade-off. One more interesting future work can be to study the power-performance trade-off by building analytical model ([WL08], [HK10]) which can capture the power requirements for serial and parallel cores. There can be different conclusion on building future many-cores if we consider power requirements of serial and parallel parts of the application.

Finally, this thesis work has led to 3 publications as listed below

1. **"An Empirical High level performance model for future many-cores"**, *ACM International Conference on Computing Frontiers 2015.*
2. **"Sequential and parallel code sections are different: they may require**

different processors" ,*HiPEAC - 2015, PARMA-DITAM* .

3. **" Impact of serial scaling of multi-threaded programs in many-core era"**,*SBAC-PADW - 2014*.

Appendix A

Appendix

In this section, we present the extra results obtained for the Xeon Westmere processor. As mentioned in Chapter 3, we show our SPS model for few large many-core processor and validate the model here. The explanation for understanding the model parameters remains same like in Chapter 3 and the serial section model of Xeon is used for demonstrating the benefit of heterogeneous model using large core for serial section from Xeon and many small cores from xeon-phi for parallel section. There is a slight

	serial section	Parallel section
can	$5726.89I^{-0.005}P^{-0.009}$	$14005I^{0.961}P^{-0.842}$
swap	0	$8362I^{1.02}P^{-0.9835}$
body	$553.2I^{0.9646}P^{0.047}$	$14519I^{1.01}P^{-0.982}$
fluid	$1013.9I^{0.0172}P^{-0.09}$	$2372.8I^{0.98}P^{-0.737}$
deltri	$1396I^{0.884}P^{-0.02}$	$1847.5I^{0.9911}P^{-0.786}$
preflow	$26.78I^{0.985}P^{0.007}$	$3200.3I^{1.0497}P^{-0.923}$
boruvka	$456.04I^{0.988}P^{0.008}$	$11247I^{1.061}P^{-0.936}$
barneshut	$54.4I^{1.014}P^{-0.012}$	$6187.5I^{1.964}P^{-0.971}$
survey	$454I^{1.02}P^{0.002}$	$16487I^{1.1}P^{-0.949}$

Table A.1: SPS parameters for different applications executed on Xeon Westmere obtained using trainingset$\{I \leq 16, P \leq 16\}$

difference between the applications used in both the processors due to troubles with building the application in the platform. *Body* was not buildable in Xeon-phi and *sssp* was not buildable in Xeon.

Table A.1 shows the model obtained for Xeon processors using the training set

$\{I \leq 16, P \leq 16\}$. Figure A.1 shows the validation of the model in Xeon processor with the validation set {I=32,P $\leq$ 24}[1]. The prediction error range for every application is illustrated by using a boxplot that shows the minimum, maximum and median percentage errors in these applications. We can observe from Figure A.1 that the median error for predicting the application performance running 24 threads are in the range of 7 to -3 percentage which can be acceptable for our high level modeling as SPS model gives upper bound of the achievable performance.

Table A.2 compares the achievable speedup for the considered application when different models are used. Between Xeon and Xeon-Phi, we can notice a great difference in the performance of *swap* when executed in 1024 cores $speedup_{phi} = 174$ and $speedup_{xeon} = 912$ in the SPS model. This is mainly because of the parallel scaling factor $PPS_{phi} = -0.744$ and $PPS_{xeon} = -0.983$. So in this case, few big core many-core processors seems better.

Table A.3 shows how the parallel fraction f varies as we increase the input set size I in the Xeon platform. Comparing it with Xeon-phi, we infer that the parallel fraction f of an application running may vary depending on the underlying architecture and the input set size used. For example, *deltri* running on Xeon phi has a parallel fraction $f = 0.959$ and one running on Xeon has a parallel fraction $f = 0.570$.

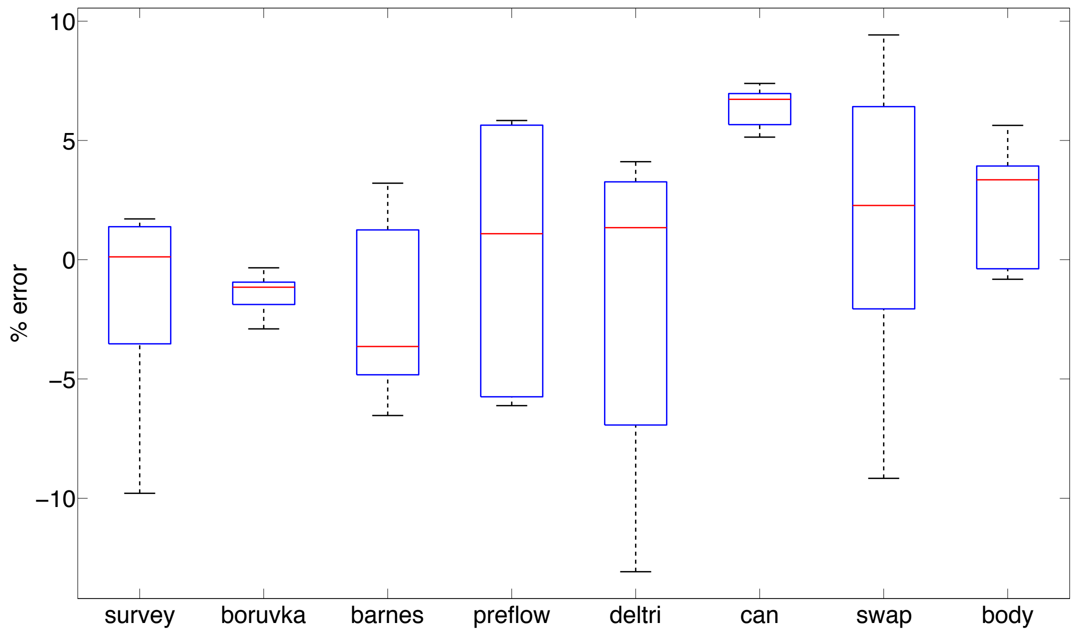


Figure A.1: Box plot showing error in few large many-core Xeon

[1]Fluid is not included in validation because it executes only with 2^n threads

Benchmark	f	Amdhal	Gustafson	GSSE	SPS
can	0.995	175.26	1019.16	888.86	131.10
swap	1.000	1022.80	1024.00	1023.96	912.50
body	0.971	33.49	994.43	524.93	33.35
deltri	0.684	3.16	700.98	65.88	3.14
preflow	0.994	138.07	1017.58	852.10	126.00
boruvka	0.981	51.02	1004.93	637.27	49.66
barneshut	1.000	919.33	1023.89	1020.37	768.20
survey	0.980	47.96	1003.65	621.18	24.00

Table A.2: Achievable speedup for the applications with $I = 100$ and $P = 1024$ in Xeon.

Benchmark	1	10	100	1000	10000
can	0.710	0.958	0.995	0.999	1.000
swap	1.000	1.000	1.000	1.000	1.000
body	0.965	0.968	0.971	0.974	0.976
deltri	0.570	0.629	0.684	0.735	0.780
preflow	0.992	0.993	0.994	0.995	0.995
boruvka	0.961	0.973	0.981	0.987	0.991
barneshut	0.991	0.999	1.000	1.000	1.000
survey	0.973	0.977	0.980	0.983	0.985

Table A.3: Parallel fraction f for varying Input set size from I=1 to 10000 for applications executed on Xeon.

Bibliography

[AAFM10] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *ACM Sigplan Notices*, volume 45, pages 739–753. ACM, 2010.

[ABC$^+$06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, Katherine A. Yelick, Meetings Jim Demmel, William Plishker, John Shalf, Samuel Williams, and Katherine Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, TECHNICAL REPORT, UC BERKELEY, 2006.

[AEJK$^+$09] Dennis Abts, Natalie D. Enright Jerger, John Kim, Dan Gibson, and Mikko H. Lipasti. Achieving predictable performance through better memory controller placement in many-core cmps. *SIGARCH Comput. Archit. News*, 37(3):451–461, June 2009.

[AGS05] M. Annavaram, E. Grochowski, and J. Shen. Mitigating amdahl's law through epi throttling. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 298–309, June 2005.

[AL07] A. Agarwal and M. Levy. The kill rule for multicore. In *Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE*, pages 750–753, June 2007.

[ALE02] Todd Austin, Eric Larson, and Dan Ernst. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[AML$^+$10] Omid Azizi, Aqeel Mahesri, Benjamin C. Lee, Sanjay J. Patel, and Mark Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis. *SIGARCH Comput. Archit. News*, 38(3):26–36, June 2010.

[ANM$^+$12] M. Arora, S. Nath, S. Mazumdar, S.B. Baden, and D.M. Tullsen. Redefining the role of the cpu in the era of cpu-gpu integration. *Micro, IEEE*, 32(6):4–16, Nov 2012.

[Ant13] Sebastian Anthony. Intel unveils 72-core x86 knights landing cpu for exascale supercomputing, 2013.

[BDK$^+$] Shekhar Y. Borkar, Pradeep Dubey, Kevin C. Kahn, David J. Kuck, Hans Mulder, Edited R. M. Ramanathan, Vince Thomas, Intel Corporation, and Stephen S. Pawlowski. Intel processor and platform evolution for the next decade executive summary.

[BEE12] M. Breughe, S. Eyerman, and L. Eeckhout. A mechanistic performance model for superscalar in-order processors. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 14–24, April 2012.

[BEE15] Maximilien B. Breughe, Stijn Eyerman, and Lieven Eeckhout. Mechanistic analytical modeling of superscalar in-order processor performance. *ACM Trans. Archit. Code Optim.*, 11(4):50:1–50:26, January 2015.

[BFH$^+$04] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: Stream computing on graphics hardware. In *ACM SIGGRAPH 2004 Papers*, SIGGRAPH '04, pages 777–786, New York, NY, USA, 2004. ACM.

[BJK$^+$95] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[BKA10] Ali Bakhoda, John Kim, and Tor M. Aamodt. Throughput-effective on-chip networks for manycore accelerators. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 421–432, Washington, DC, USA, 2010. IEEE Computer Society.

[BKSL08] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[BL10] C. Bienia and K. Li. Fidelity and scaling of the parsec benchmark inputs. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, pages 1–10, 2010.

[Bor99] S. Borkar. Design challenges of technology scaling. *Micro, IEEE*, 19(4):23–29, Jul 1999.

[Bor07] Shekhar Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[BPBL06] Pieter Bellens, Josep M Perez, Rosa M Badia, and Jesus Labarta. Cellss: a programming model for the cell be architecture. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 5–5. IEEE, 2006.

[BPW+95] D.R. Beard, A.E. Phelps, M.A. Woodmansee, R.G. Blewett, J.A. Lohman, A.A. Silbey, G.A. Spix, F.J. Simmons, and D.A. Van Dyke. Scalar/vector processor, July 4 1995. US Patent 5,430,884.

[BRL+08] Bradley J. Barnes, Barry Rountree, David K. Lowenthal, Jaxk Reeves, Bronis de Supinski, and Martin Schulz. A regression-based approach to scalability prediction. In *Proceedings of the 22Nd Annual International Conference on Supercomputing*, ICS '08, pages 368–377, New York, NY, USA, 2008. ACM.

[BRUL05] S. Balakrishnan, R. Rajwar, M. Upton, and Konrad Lai. The impact of performance asymmetry in emerging multicore architectures. In *Computer Architecture, 2005. ISCA '05. Proceedings. 32nd International Symposium on*, pages 506–517, June 2005.

[BSW+00] J Mark Bull, Lorna A Smith, Martin D Westhead, David S Henty, and Robert A Davey. A benchmark suite for high performance java. *Concurrency - Practice and Experience*, 12(6):375–388, 2000.

[BWFM09] N. Barrow-Williams, C. Fensch, and S. Moore. A communication characterisation of splash-2 and parsec. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 86–97, Oct 2009.

[CB92] Tien-Fu Chen and Jean-Loup Baer. Reducing memory latency via non-blocking and prefetching caches. *SIGPLAN Not.*, 27(9):51–61, September 1992.

[CBM+09] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[CG95] Thomas M Conte and Charles E Gimarc. *Fast simulation of computer architectures.* Springer, 1995.

[chr] Microprocessor chronology, http://en.wikipedia.org/wiki/microprocessor_chronology.

[CJRP14] Villavieja Carlos, A. Joao Jose, Miftakhutdinov Rustam, and Yale N. Patt. Yoga: A hybrid dynamic vliw/ooo processor. Technical report, High Performance Systems Group, March 2014.

[CK94] Bob Cmelik and David Keppel. Shade: A fast instruction-set simulator for execution profiling. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '94, pages 128–137, New York, NY, USA, 1994. ACM.

[cou] Transistor count, http://en.wikipedia.org/wiki/transistor_count.

[CZHG06] Juan Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Toward a software infrastructure for the cyclops-64 cellular architecture. In *In the 20th International Symposium on High Performance Computing Systems and Applications (HPCS2006), St. John Rs, Newfoundland and*, 2006.

[Dal06] B. Dally. Computer architecture in the many-core era. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 1–1, Oct 2006.

[DGR$^+$74] Robert H Dennard, Fritz H Gaensslen, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, 1974.

[DJO07] C. Dubach, T.M. Jones, and M.F.P. O'Boyle. Microarchitectural design space exploration using an architecture-centric approach. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 262–271, Dec 2007.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 1998.

[dM10] Arnaldo Carvalho de Melo. The new linux perf tools. In *Slides from Linux Kongress*, 2010.

[DS11] John Demme and Simha Sethumadhavan. Rapid identification of architectural bottlenecks via precise event counting. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 353–364. ACM, 2011.

[DTC08] Paul J Drongowski, AMD CodeAnalyst Team, and Boston Design Center. An introduction to analysis and optimization with amd codeanalyst performance analyzer. *Advanced Micro Devices, Inc*, 2008.

[EBJS$^+$04] Lieven Eeckhout, Robert H Bell Jr, Bastiaan Stougie, Koen De Bosschere, and Lizy K John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 350–361. IEEE, 2004.

[EBSA$^+$11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[EDBE12] Stijn Eyerman, Kristof Du Bois, and Lieven Eeckhout. Speedup stacks: Identifying scaling bottlenecks in multi-threaded applications. In *Performance Analysis of Systems and Software (ISPASS), 2012 IEEE International Symposium on*, pages 145–155. IEEE, 2012.

[Edw] Stephen A. Edwards. http://www.cs.columbia.edu/ sedwards/classes/2012/3827-spring/advanced-arch-2011.pdf.

[EE10] Stijn Eyerman and Lieven Eeckhout. Modeling critical sections in amdahl's law and its implications for multicore design. In *Conference Proceedings Annual International Symposium on Computer Architecture*, pages 362–370. Association for Computing Machinery (ACM), 2010.

[EEKS09] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E. Smith. A mechanistic performance model for superscalar out-of-order processors. *ACM Trans. Comput. Syst.*, 27(2):3:1–3:37, May 2009.

[EH] J Edler and M. D Hill. Dinero iv: trace-driven uniprocessor cache simulator. Technical report.

[Era03] Stephane Eranian. Perfmon: Linux performance monitoring for ia-64. *Downloadable software with documentation, http://www. hpl. hp. com/research/linux/perfmon*, 2003.

[FBJ$^+$08] Nadeem Firasta, Mark Buxton, Paula Jinbo, Kaveh Nasri, and Shihjong Kuo. Intel avx: New frontiers in performance improvements and energy efficiency. *Intel white paper*, 2008.

[FHK$^+$06] Kayvon Fatahalian, Daniel Reiter Horn, Timothy J Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J Dally, et al. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83. ACM, 2006.

[FO84] Joseph A Fisher and John J O'Donnell. Vliw machines: Multiprocessors we can acutally program. In *CompCon*, pages 299–305, 1984.

[FTP94] M. Farrens, G. Tyson, and A.R. Pleszkun. A study of single-chip processor/cache organizations for large numbers of transistors. In *Computer Architecture, 1994., Proceedings the 21st Annual International Symposium on*, pages 338–347, Apr 1994.

[GHG+91] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, ISCA '91, pages 254–263, New York, NY, USA, 1991. ACM.

[GKS12] Vishal Gupta, Hyesoon Kim, and Karsten Schwan. Evaluating scalability of multi-threaded applications on a many-core platform. Technical report, Georgia Institute of Technology, 2012.

[GLS99] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[GRE+01] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[GU96] W. Grunewald and T. Ungerer. Towards extremely fast context switching in a block-multithreaded processor. In *EUROMICRO 96. Beyond 2000: Hardware and Software Design Strategies., Proceedings of the 22nd EUROMICRO Conference*, pages 592–599, Sep 1996.

[Gus88] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988.

[Gwe11] Linley Gwennap. Adapteva: More flops, less watts. 2011.

[HAP+05] M. Horowitz, E. Alon, D. Patil, S. Naffziger, Rajesh Kumar, and K. Bernstein. Scaling, power, and the future of cmos. In *Electron Devices Meeting, 2005. IEDM Technical Digest. IEEE International*, pages 7 pp.–15, Dec 2005.

[HBK01] Jaehyuk Huh, D. Burger, and S.W. Keckler. Exploring the design space of future cmps. In *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, pages 199–210, 2001.

[HBT13] Tomas Hruby, Herbert Bos, and Andrew S. Tanenbaum. When slower is faster: On heterogeneous multicores for reliable systems. In *Proceedings of USENIX ATC*, San Jose, CA, USA, June 2013. USENIX, USENIX.

[HDH+10] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen,

S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108–109, Feb 2010.

[HE07] K. Hoste and L. Eeckhout. Microarchitecture-independent workload characterization. *Micro, IEEE*, 27(3):63–72, May 2007.

[HK10] Sunpyo Hong and Hyesoon Kim. An integrated gpu power and performance model. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 280–289. ACM, 2010.

[HKO$^+$14] Per Hammarlund, Rajesh Kumar, Randy B Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, et al. Haswell: The fourth-generation intel core processor. *IEEE Micro*, (2):6–20, 2014.

[HM08] Mark D Hill and Michael R Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[HP03] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3 edition, 2003.

[IdSSM05] Engin Ipek, Bronis R. de Supinski, Martin Schulz, and Sally A. McKee. An approach to performance prediction for parallel applications. In *Proceedings of the 11th International Euro-Par Conference on Parallel Processing*, Euro-Par'05, pages 196–205, Berlin, Heidelberg, 2005. Springer-Verlag.

[IKKM07] Engin Ipek, Meyrem Kirman, Nevin Kirman, and Jose F. Martinez. Core fusion: Accommodating software diversity in chip multiprocessors. *SIGARCH Comput. Archit. News*, 35(2):186–197, June 2007.

[IMC$^+$06] Engin Ipek, Sally A. McKee, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. Efficiently exploring architectural design spaces via predictive modeling. *SIGPLAN Not.*, 41(11):195–206, October 2006.

[int] Intel e7 8890 v2, online at: http://ark.intel.com/products/75258/intel-xeon-processor-e7-8890-v2-37_5m-cache-2_80-ghz.

[ITR15] International technology roadmap for semiconductors, april 2015.

[Jef12] Brian Jeff. Big.little system architecture from arm: saving power through heterogeneous multiprocessing and task context migration. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *DAC*, pages 1143–1146. ACM, 2012.

[JFY99] Hao-Qiang Jin, Michael Frumkin, and Jerry Yan. The openmp implementation of nas parallel benchmarks and its performance. 1999.

[JM12] B.H.H. Juurlink and C. H. Meenderinck. Amdahl's law for predicting the future of multicores considered harmful. *SIGARCH Comput. Archit. News*, 40(2):1–9, May 2012.

[Joh89] W. M. Johnson. *Super-scalar Processor Design.* PhD thesis, Stanford, CA, USA, 1989. UMI Order No: GAX89-25892.

[JR13] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

[JVT06a] PJ Joseph, Kapil Vaswani, and Matthew J Thazhuthaveetil. Construction and use of linear regression models for processor performance analysis. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 99–108. IEEE, 2006.

[JVT06b] PJ Joseph, Kapil Vaswani, and Matthew J Thazhuthaveetil. A predictive performance model for superscalar processors. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 161–170. IEEE Computer Society, 2006.

[K$^+$07] David Kirk et al. Nvidia cuda software and gpu parallel computing architecture. In *ISMM*, volume 7, pages 103–104, 2007.

[Kal13] MPPA Kalray. 256 many-core processor, 2013.

[KBCP09] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. Lonestar: A suite of parallel irregular programs. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 65–76. IEEE, 2009.

[KBI$^+$09] Milind Kulkarni, Martin Burtscher, Rajeshkar Inkulu, Keshav Pingali, and Calin Casçaval. How much parallelism is there in irregular applications? *SIGPLAN Not.*, 44(4):3–14, February 2009.

[KFJ$^+$03] R. Kumar, K.I. Farkas, N.P. Jouppi, P. Ranganathan, and D.M. Tullsen. Single-isa heterogeneous multi-core architectures: the potential for processor power reduction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 81–92, Dec 2003.

[KPW$^+$07] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L Paul Chew. Optimistic parallelism requires abstractions. In *ACM SIGPLAN Notices*, volume 42, pages 211–222. ACM, 2007.

[KS04] Tejas S. Karkhanis and James E. Smith. A first-order superscalar processor model. *SIGARCH Comput. Archit. News*, 32(2):338–, March 2004.

[KSG$^+$07] Changkyu Kim, Simha Sethumadhavan, Madhu S Govindan, Nitya Ranganathan, Divya Gulati, Doug Burger, and Stephen W Keckler. Composable lightweight processors. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 381–394. IEEE, 2007.

[KSH$^+$12] K. Khubaib, M.A. Suleman, M. Hashemi, C. Wilkerson, and Y.N. Patt. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 305–316, Dec 2012.

[KSSF10] Ron Kalla, Balaram Sinharoy, William J. Starke, and Michael Floyd. Power7: Ibm's next-generation server processor. *IEEE Micro*, 30(2):7–15, March 2010.

[KTJ06] Rakesh Kumar, Dean M. Tullsen, and Norman P. Jouppi. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, PACT '06, pages 23–32, New York, NY, USA, 2006. ACM.

[Lar90] James R Larus. Abstract execution: A technique for efficiently tracing programs. *Software: Practice and Experience*, 20(12):1241–1258, 1990.

[LB06] Benjamin C Lee and David M Brooks. Accurate and efficient regression modeling for microarchitectural performance and power prediction. In *ACM SIGPLAN Notices*, volume 41, pages 185–194. ACM, 2006.

[LBdS$^+$07] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '07, pages 249–258, New York, NY, USA, 2007. ACM.

[LCM$^+$05] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[LDT$^+$12] Jean-Pierre Lozi, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 6–6, Berkeley, CA, USA, 2012. USENIX Association.

[Lev04] John Levon. Oprofile manual. *Victoria University of Manchester*, 2004.

[LKV12] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn&rsquo;t, and why. *ACM Trans. Archit. Code Optim.*, 9(1):2:1–2:29, March 2012.

[LM98] Chi-Keung Luk and Todd C. Mowry. Cooperative prefetching: Compiler and hardware support for effective instruction prefetching in modern processors. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 31, pages 182–194, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[LNOM08] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, March 2008.

[LÖ09] Ling Liu and M. Tamer Özsu, editors. *Encyclopedia of Database Systems.* Springer US, 2009.

[Lom11] Chris Lomont. Introduction to intel advanced vector extensions. *Intel White Paper*, 2011.

[LPD$^+$12] Andrew Lukefahr, Shruti Padmanabha, Reetuparna Das, Faissal M Sleiman, Ronald Dreslinski, Thomas F Wenisch, and Scott Mahlke. Composite cores: Pushing heterogeneity into a core. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 317–328. IEEE Computer Society, 2012.

[LPMS97] Chunho Lee, Miodrag Potkonjak, and William H Mangione-Smith. Mediabench: a tool for evaluating and synthesizing multimedia and communicatons systems. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 330–335. IEEE Computer Society, 1997.

[LW92] Monica S. Lam and Robert P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, pages 46–57, New York, NY, USA, 1992. ACM.

[Mar11] Ami Marowka. On performance analysis of a multithreaded application parallelized by different programming models using intel vtune. In *Proceedings of the 11th International Conference on Parallel Computing Technologies*, PaCT'11, pages 317–331, Berlin, Heidelberg, 2011. Springer-Verlag.

[MBDH99] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. Papi: A portable interface to hardware performance counters. In *Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[MBS+11] A.D. Malony, S. Biersdorff, S. Shende, H. Jagode, S. Tomov, G. Juckeland, R. Dietrich, D. Poole, and C. Lamb. Parallel performance measurement of heterogeneous parallel systems with gpus. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 176–185, Sept 2011.

[MCE+02] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.

[McF93] Scott McFarling. Combining branch predictors. Technical report, Technical Report TN-36, Digital Western Research Laboratory, 1993.

[MFZY14] Sanyam Mehta, Zhenman Fang, Antonia Zhai, and Pen-Chung Yew. Multi-stage coordinated prefetching for present-day processors. In *Proceedings of the 28th ACM international conference on Supercomputing*, pages 73–82. ACM, 2014.

[MIP] Instructions per second, http://en.wikipedia.org/wiki/instructions_per_second#timeline_

[MJS11] M. Manivannan, B. Juurlink, and P. Stenstrom. Implications of merging phases on scalability of multi-core architectures. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 622–631, 2011.

[Moo98] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.

[mor] Morphmark midlet, online at: http://www.morpheme.co.uk/.

[MSJ99] Pierre Michaud, Andre Seznec, and Stephan Jourdan. Exploring instruction-fetch bandwidth requirement in wide-issue superscalar processors. In *IN PROCEEDINGS OF THE INTERNATIONAL CONFERENCE ON PARALLEL ARCHITECTURES AND COMPILATION TECHNIQUES*, pages 2–10, 1999.

[NBF98] Bradford Nichols, Dick Buttlar, and Jacqueline P Farrell. Pthreads programming. 1998.

[NBS14] S.N. Natarajan, .N.S Bharath, and Andre Seznec. Impact of serial scaling of multi-threaded programs in many-core era. *5th Workshop on Applications for Multi-Core Architectures, SBAC-PAD*, 2014.

[NL09] Van Hoa Nguyen and Dominique Lavenier. Plast: parallel local alignment search tool for database comparison. *BMC Bioinformatics*, 10:329, 2009.

[PDG06] Jeff Parkhurst, John Darringer, and Bill Grundmann. From single core to multi-core: preparing for a new exponential. In *Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, ICCAD '06, pages 67–72, New York, NY, USA, 2006. ACM.

[Phe08] Chuck Pheatt. Intel threading building blocks. *Journal of Computing Sciences in Colleges*, 23(4):298–298, 2008.

[PJJ07] Aashish Phansalkar, Ajay Joshi, and Lizy K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. *SIGARCH Comput. Archit. News*, 35(2):412–423, June 2007.

[PJS97] Subbarao Palacharla, Norman P. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. *SIGARCH Comput. Archit. News*, 25(2):206–218, May 1997.

[Pol99] Fred J. Pollack. New microarchitecture challenges in the coming generations of cmos process technologies. *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 0:2, 1999.

[Poo07] Jason Poovey. Characterization of the eembc benchmark suite. *North Carolina State University*, 2007.

[PS13] Arthur Perais and André Seznec. EOLE: Paving the Way for an Effective Implementation of Value Prediction. Research Report RR-8402, November 2013. A fait l'objet d'une publication au "International Symposium on Computer Architecture (ISCA) 2014" Lien : http://people.irisa.fr/Arthur.Perais/data/ISCA

[PS15] A. Perais and A. Seznec. Bebop: A cost effective predictor infrastructure for superscalar value prediction. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*, pages 13–25, Feb 2015.

[PSM95] Jim Pierce, Michael D Smith, and Trevor Mudge. Instrumentation tools. In *Fast Simulation of Computer Architectures*, pages 47–86. Springer, 1995.

[PW96] A. Peleg and U. Weiser. Mmx technology extension to the intel architecture. *Micro, IEEE*, 16(4):42–50, Aug 1996.

[Rei05] James Reinders. *VTune performance analyzer essentials*. Intel Press, 2005.

[RHWG95] Mendel Rosenblum, Stephen A Herrod, Emmett Witchel, and Anoop Gupta. Complete computer system simulation: The simos approach. *Parallel & Distributed Technology: Systems & Applications, IEEE*, 3(4):34–43, 1995.

[Roh11] Erven Rohou. Tiptop: Hardware performance counters for the masses. Rapport de recherche RR-7789, INRIA, November 2011.

[RWJ09] Shah Mohammad Faizur Rahman, Zhe Wang, and Daniel A. Jiménez. Studying microarchitectural structures with object code reordering. In

Proceedings of the Workshop on Binary Instrumentation and Applications, WBIA '09, pages 7–16, New York, NY, USA, 2009. ACM.

[SC02] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 25–34, 2002.

[SE04] Amitabh Srivastava and Alan Eustace. Atom: A system for building customized program analysis tools. *ACM SIGPLAN Notices*, 39(4):528–539, 2004.

[Sez10] Andre Seznec. Defying amdahl's law - dal. Technical report, INRIA, 2010.

[Sez11] André Seznec. A new case for the tage branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, pages 117–127, New York, NY, USA, 2011. ACM.

[SHU$^+$01] D. Sager, G. Hinton, M. Upton, T. Chappell, T.D. Fletcher, S. Samaan, and R. Murray. A 0.18 /spl mu/m cmos ia32 microprocessor with a 4 ghz integer execution unit. In *Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC. 2001 IEEE International*, pages 324–325, Feb 2001.

[SKM$^+$06] Srinivas Sridharan, Brett Keck, Richard Murphy, Surendar Chandra, and Peter Kogge. Thread migration to improve synchronization performance. In *Workshop on Operating System Interference in High Performance Applications*, volume 34, page 35, 2006.

[SKS14] B.N. Swamy, A. Ketterlin, and A. Seznec. Hardware/software helper thread prefetching on heterogeneous many cores. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, pages 214–221, Oct 2014.

[Smi81] James E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 135–148, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[SMQP09] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. *SIGPLAN Not.*, 44(3):253–264, March 2009.

[SMYH12] Toshinori Sato, Hideki Mori, Rikiya Yano, and Takanori Hayashida. Importance of single-core performance in the multicore era. In *Proceedings of the Thirty-fifth Australasian Computer Science Conference-Volume 122*, pages 107–114. Australian Computer Society, Inc., 2012.

[Spr02] B. Sprunt. The basics of performance-monitoring hardware. *Micro, IEEE*, 22(4):64–71, Jul 2002.

[STH$^+$10] J.L. Shin, K. Tam, D. Huang, B. Petrick, H. Pham, Changku Hwang, Hongping Li, A. Smith, T. Johnson, F. Schumacher, D. Greenhill, A.S. Leon, and A. Strong. A 40nm 16-core 128-thread cmt sparc soc processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 98–99, Feb 2010.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

[TEL95] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*, pages 392–403, June 1995.

[THW10] J. Treibig, G. Hager, and G. Wellein. Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, pages 207–216, Sept 2010.

[til] Tilera chip, online at: http://www.tilera.com/products/?ezchip=585&spage=618.

[TPC] Transactions processing council, online at: http://www.tpc.org.

[Uni89] Joseph Uniejewski. Spec benchmark suite: designed for today's advanced systems. *SPEC Newsletter*, 1(1):1, 1989.

[VHvN$^+$] Ana Lucia Varbanescu, Pieter Hijma, Rob V van Nieuwpoort, Henri E Bal, Rosa M Badia, and Xavier Martorell. Programming models for many-cores.

[VNGE$^+$07] Ana Lucia Varbanescu, Maik Nijhuis, Arturo González-Escribano, Henk Sips, Herbert Bos, and Henri Bal. Sp@ ce-an sp-based programming model for consumer electronics streaming applications. In *Languages and Compilers for Parallel Computing*, pages 33–48. Springer, 2007.

[VSG$^+$10] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: Reducing the energy of mature computations. *SIGARCH Comput. Archit. News*, 38(1):205–218, March 2010.

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 176–188, New York, NY, USA, 1991. ACM.

[WBM$^+$03] Zhenlin Wang, Doug Burger, Kathryn S. McKinley, Steven K. Reinhardt, and Charles C. Weems. Guided region prefetching: A cooperative hardware/software approach. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, ISCA '03, pages 388–398, New York, NY, USA, 2003. ACM.

[WL08] Dong Hyuk Woo and Hsien-Hsin S. Lee. Extending amdahl's law for energy-efficient computing in the many-core era. *Computer*, 41(12):24–31, 2008.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, March 1995.

[Wol04] Alexandre Wolfe. Intel clears up post tejas confusion, online at : http://www.crn.com/news/channel-programs/18842588/intel-clears-up-post-tejas-confusion.htm, 2004.

[WOT$^+$95] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36. ACM, 1995.

[YMA$^+$13] Toshio Yoshida, Takumi Maruyama, Yasunobu Akizuki, Ryuji Kan, Naohiro Kiyota, Kiyoshi Ikenishi, Shigeki Itou, Tomoyuki Watahiki, and Hiroshi Okano. Sparc64 x: Fujitsu's new-generation 16-core processor for unix servers. *IEEE Micro*, 33(6):16–24, 2013.

[YMG14] Leonid Yavits, Amir Morad, and Ran Ginosar. The effect of communication and synchronization on amdahl's law in multicore systems. *Parallel Computing*, 40(1):1–16, 2014.

[YP91] Tse-Yu Yeh and Yale N. Patt. Two-level adaptive training branch prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, MICRO 24, pages 51–61, New York, NY, USA, 1991. ACM.

[ZC] Huiyang Zhou and T Conte. Performance modeling of memory latency hiding techniques. Technical report.

List of Figures