Universidade Estadual de Campinas
Instituto de Computação

Martin Ichilevici de Oliveira

PTB: An Integrated Page, Thread and Bandwidth Allocation Approach for NUMA Architectures

PTB: Uma Abordagem Integrada de Alocação de Páginas, Threads e Banda para Arquiteturas NUMA

CAMPINAS
2016

Martin Ichilevici de Oliveira

PTB: An Integrated Page, Thread and Bandwidth Allocation Approach for NUMA Architectures

PTB: Uma Abordagem Integrada de Alocação de Páginas, Threads e Banda para Arquiteturas NUMA

Dissertação apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Computer Science.

Supervisor/Orientador: Prof. Dr. Guido Costa Souza de Araújo
Co-supervisores/Coorientadores: Prof. Dr. Alexandro José Baldassin
Prof. Dr. José Nelson Amaral

Este exemplar corresponde à versão da Dissertação entregue à banca antes da defesa.

CAMPINAS
2016

Acknowledgements

First and foremost, I would like to thank my beloved wife Ana, whose companionship, support and understanding are greatly treasured. I would also like to thank my mother, Vita, for encouraging me to pursue an academic career and my father, Gustavo, for introducing me to math and science so early and for keeping it interesting and fun at all times. I would like to thank my brother Alan and my sister Lara for their friendship and confidence.

I would also like to thank my advisor, Guido Araújo, for all the incentive and patience throughout our project. I am very grateful for his openness and guidance ever since I approached him to learn about a research career.

I thank my co-advisor Alexandro Baldassin for always helping me think outside the box. I am also thankful to my co-advisor, Nelson Amaral, who taught me so about being a researcher and for welcoming me in Canada. I thank Emílio Francesquini for his support and all the effort he put into this work.

I would like to thank the financial support of CAPES and FAPESP.

Resumo

A correta alocação de *threads* e páginas de memória em nós de computadores de uma arquitetura NUMA (*Non-Uniform Memory Access*), visando igualar a demanda por acessos remotos com a disponibilidade de banda de comunicação entre os nós, pode ter um impacto significativo no tempo de execução de programas. Tipicamente, esta alocação deve atender quatro objetivos simultaneamente: (a) manter *threads* próximas às páginas que elas acessam; (b) distribuir a carga de trabalho de maneira uniforme entre todos os nós; (c) manter a demanda à memória abaixo da banda suportada pelo controlador de memória de cada nó; e (d) realocar *threads* e páginas de memória de acordo com alterações nos padrões de acesso do programa. A maioria das soluções existentes para o problema de distribuição de *threads* e páginas em máquinas NUMA concentra-se somente em um subconjunto destas metas, principalmente devido à complexidade das soluções requeridas e à sobrecarga resultante da implementação.

Este trabalho propõe o algoritmo PTB, uma heurística que busca simultaneamente alocar (P)áginas, (T)hreads e (B)anda por todos os nós de uma arquitetura NUMA. Ao contrário de abordagens alternativas, a solução integrada PTB procura, ao mesmo tempo, distribuir o trabalho de maneira uniforme entre todos os nós e limitar a demanda aos controladores de memória de cada nó. Além disso, o algoritmo aborda questões de assimetria presentes na infraestrutura de comunicação de máquinas NUMA modernas.

Resultados experimentais utilizando *benchmarks* das suítes Parsec, NAS e Metis revelam que PTB produz média geométrica de *speedup* 1.16x em relação ao escalonador padrão do Linux. Além disso, para um grande conjunto de programas, PTB produz *speedups* entre 1.6x e 2.0x, enquanto a solução para balanceamento NUMA do Linux mantém-se abaixo de 1.2x ou até mesmo os desacelera.

Abstract

In a NUMA machine, a program's execution time can be significantly impacted by how data and tasks are distributed between nodes. Thus, correctly assigning threads and memory pages is paramount. The correct assignment should match the demand for remote data transfers with the available communication bandwidth and memory controller capacity.

Such assignment typically requires dealing with four simultaneous goals: (a) keep threads close to the memory pages they access; (b) evenly distribute the workload among nodes; (c) maintain memory demand below memory controllers' bandwidth; and (d) reassign threads and pages to follow changes in the memory access pattern of the program. However, most solutions to this problem address only a subset of these goals, mainly because they seek to avoid complex solutions or expensive implementation overheads.

This work proposes PTB, a heuristic-based algorithm that simultaneously allocates (P)ages, (T)hreads and (B)andwidth to each node of a NUMA architecture. In contrast to alternative approaches, PTB integrated solution seeks both to uniformly distribute workload and to limit memory demand to the controllers' bandwidth while also addressing asymmetry issues found in the communication paths of modern NUMA architectures.

Experimental results using Parsec, NAS and Metis benchmarks reveal that PTB produces geometric mean speedups of 1.16x when compared to Linux's default scheduler. In particular, for a number of programs, PTB speedups ranged from 1.6x to 2x while Linux's automatic NUMA balancing either stayed below 1.2x. or resulted in slowdowns.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Non-Uniform Memory Access (NUMA) architectures are machines containing a large number of cores organized in *nodes*; each node has its own physical memory and nodes are linked together using a fast interconnect network [46]. Although physical memory is distributed among nodes, NUMA hardware and OS provide a mechanism that enables a single memory address space programming model. This mechanism detects the node where the accessed data is located and forwards the access to that node's physical memory. Given that accesses to the memory of a local node (*local memory*) are faster than those to a remote node memory (*remote memory*), assigning threads and memory pages to NUMA nodes in a way that leads to reduction in program execution time is an important research problem.

A solution to this problem requires a careful evaluation of the tradeoff among four conflicting goals. First, a thread should be allocated to the node that contains the pages that the thread accesses so that it can benefit from fast local memory accesses. Second, given that typical parallel programs share data, it is often not possible to both evenly distribute the thread's workload amongst the nodes and keep all accesses local; hence the allocation algorithm has to determine which pages should stay local and which pages will be accessed remotely. Third, memory controllers have limited bandwidth and thus the algorithm must distribute pages in a way that does not exceed the memory throughput of each controller; if the memory demand for a specific node goes above its memory controller supported bandwidth, memory accesses are delayed thus reducing performance. Finally, programs have execution phases; memory accesses change during a program execution and thus require the migration of pages and threads to avoid performance degradation. In general, any solution to the NUMA thread/page allocation problem requires addressing all the above mentioned goals together.

Figure 1.1, which is adapted from a NUMA machine documentation [12], shows a typical NUMA configuration with four nodes, and a partial communication interconnect. A detailed description of such architecture is available in Chapter 4. This machine exhibits asymmetric features that typically are not taken into consideration in the design of NUMA performance optimizers. For example, as shown in Figure 1.1, not all nodes are directly connected to memory, thus optimizing for locality is not enough to improve performance. Moreover, although not shown in the figure, it is not uncommon to find architectures for which the interconnection links are *asymmetric*, *i.e.* the bandwidth available for communication in each direction of a link is different [29].

Writing a NUMA-aware computer program is not an easy task. Although libraries [44], runtimes [10] and compiler support [42] have been proposed to assist developers, the allocation of threads and pages to nodes is complex and is typically left to the scheduler or operating system. Existing strategies attempt to increase data locality [45, 30, 55] and/or to minimize congestion [32, 17, 8]. For example, Linux has a NUMA-aware scheduler that seeks to co-locate threads that share data [16]. However, most of these approaches do not simultaneously take into consideration the interplay between page accesses, thread and bandwidth allocation and the existing asymmetry in the links of modern NUMA computers [29].

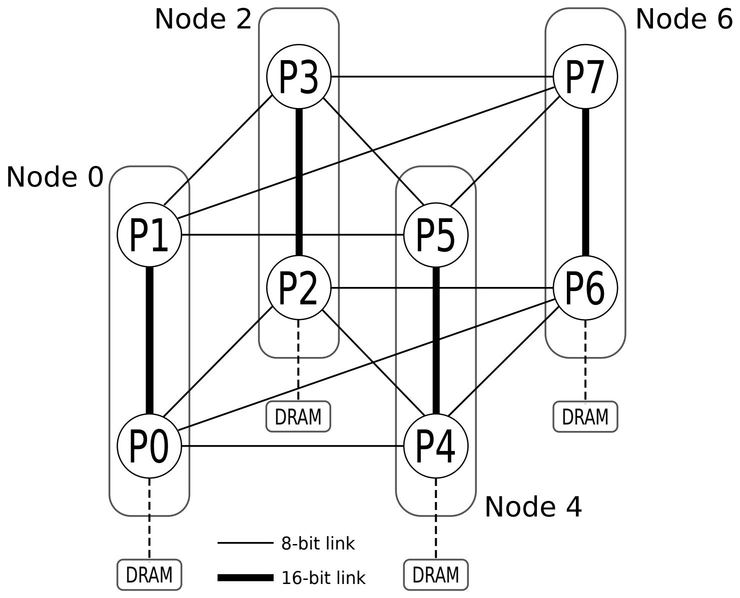


Figure 1.1: Topology of an asymmetric machine (adapted from [12])

This work proposes PTB, an approximation algorithm that simultaneously allocates (P)ages, (T)hreads and (B)andwidth to each node of a NUMA architecture. It seeks to minimize memory access latency while balancing memory bandwidth usage and taking architectural asymmetries into consideration. A key insight in the design of PTB is an empirical procedure that measures actual machine parameters such as memory-controller and interconnect bandwidths so as to build an architecture performance model. PTB uses this model, and an online execution profiling, to dynamically assign threads and pages to computing nodes. The experimental evaluation, described in Chapter 6, shows that for NPB, Parsec and Metis benchmarks PTB can result in speedups of up to 2.0x, with a geometric mean of 1.16x when compared to the default Linux NUMA-aware scheduler.

1.1 Contributions

This work makes the following contributions:

- An experimental analysis of the tradeoff between increasing the number of local accesses and reducing congestion in memory controllers. As detailed in Section 4.3, this analysis reveals that minimizing controller congestion after saturating its bandwidth is more important than prioritizing local accesses.

- A new architectural performance model of the actual hardware called *Bandwidth Graph* (BG) that summarizes the key features of a real machine. NUMA architectures can have a very large number of possible configurations resulting from the combination of: processor types, interconnect topologies, number and distribution of memory modules, interconnect bandwidth and routing algorithms, *etc.* It can be difficult for any page/thread allocation algorithm to take all these features into consideration without a model to summarize them. Section 4.2 describes how BG empirically models the architectural configuration thus abstracting the underlying hardware.

- PTB, a page, thread and bandwidth allocation algorithm for NUMA machines that uses BG to guide allocation decisions. PTB, described in Chapter 5, works in three steps that execute periodically: (a) online profiling of memory accesses performed by threads to pages; (b) calculation of thread similarity indexes, followed by the corresponding thread migrations; and (c) memory pages redistribution based on bandwidth estimation.

The rest of this work is organized as follows. Chapter 2 discusses the literature in the area putting this work in perspective. Chapter 3 explains how Linux's NUMA scheduler works. Chapter 4 describes the key features of a NUMA architecture and the machine used in this work. Chapter 5 details the PTB allocation algorithm proposed herein. Chapter 6 describes the experimental results of applying PTB to Parsec, NPB and Metis benchmarks while Chapter 7 concludes the work.

1.2 Why is this a relevant problem?

The memory system is often the bottleneck in computer systems [41]. The NUMA architecture was created and is used to minimize the overhead of memory accesses. However, a careful distribution of both data and threads between nodes is paramount in order to extract the maximum possible performance.

To illustrate why a correct placement is required, consider Figure 1.2. This figure shows the memory bandwidth of a memory-intensive application that access pseudo-random memory positions (described in further details on Section 4.1). All of this application's data is allocated in only one node (or distributed between all nodes, as in the last graph) and the experiment is repeated each time with a different thread distribution. Table 1.1 shows all the tested thread configurations used. Each of the top four graphs shows the bandwidth obtained when allocating memory exclusively in one of the four available nodes.

Table 1.1: Thread configurations used in Figure 1.2

Id	Distribution	Id	Distribution	Id	Distribution
0	0 1 2 3 4 5 6 7	36	0 1 8 9 16 17 24 25	72	8 9 16 17 24 25 40 41
1	8 9 10 11 12 13 14 15	37	0 1 8 9 16 17 32 33	73	8 9 16 17 24 25 48 49
2	16 17 18 19 20 21 22 23	38	0 1 8 9 16 17 40 41	74	8 9 16 17 24 25 56 57
3	24 25 26 27 28 29 30 31	39	0 1 8 9 16 17 48 49	75	8 9 16 17 32 33 40 41
4	32 33 34 35 36 37 38 39	40	0 1 8 9 16 17 56 57	76	8 9 16 17 32 33 48 49
5	40 41 42 43 44 45 46 47	41	0 1 8 9 24 25 32 33	77	8 9 16 17 32 33 56 57
6	48 49 50 51 52 53 54 55	42	0 1 8 9 24 25 40 41	78	8 9 16 17 40 41 48 49
7	56 57 58 59 60 61 62 63	43	0 1 8 9 24 25 48 49	79	8 9 16 17 40 41 56 57
8	0 1 2 3 8 9 10 11	44	0 1 8 9 24 25 56 57	80	8 9 16 17 48 49 56 57
9	0 1 2 3 16 17 18 19	45	0 1 8 9 32 33 40 41	81	8 9 24 25 32 33 40 41
10	0 1 2 3 24 25 26 27	46	0 1 8 9 32 33 48 49	82	8 9 24 25 32 33 48 49
11	0 1 2 3 32 33 34 35	47	0 1 8 9 32 33 56 57	83	8 9 24 25 32 33 56 57
12	0 1 2 3 40 41 42 43	48	0 1 8 9 40 41 48 49	84	8 9 24 25 40 41 48 49
13	0 1 2 3 48 49 50 51	49	0 1 8 9 40 41 56 57	85	8 9 24 25 40 41 56 57
14	0 1 2 3 56 57 58 59	50	0 1 8 9 48 49 56 57	86	8 9 24 25 48 49 56 57
15	8 9 10 11 16 17 18 19	51	0 1 16 17 24 25 32 33	87	8 9 32 33 40 41 48 49
16	8 9 10 11 24 25 26 27	52	0 1 16 17 24 25 40 41	88	8 9 32 33 40 41 56 57
17	8 9 10 11 32 33 34 35	53	0 1 16 17 24 25 48 49	89	8 9 32 33 48 49 56 57
18	8 9 10 11 40 41 42 43	54	0 1 16 17 24 25 56 57	90	8 9 40 41 48 49 56 57
19	8 9 10 11 48 49 50 51	55	0 1 16 17 32 33 40 41	91	16 17 24 25 32 33 40 41
20	8 9 10 11 56 57 58 59	56	0 1 16 17 32 33 48 49	92	16 17 24 25 32 33 48 49
21	16 17 18 19 24 25 26 27	57	0 1 16 17 32 33 56 57	93	16 17 24 25 32 33 56 57
22	16 17 18 19 32 33 34 35	58	0 1 16 17 40 41 48 49	94	16 17 24 25 40 41 48 49
23	16 17 18 19 40 41 42 43	59	0 1 16 17 40 41 56 57	95	16 17 24 25 40 41 56 57
24	16 17 18 19 48 49 50 51	60	0 1 16 17 48 49 56 57	96	16 17 24 25 48 49 56 57
25	16 17 18 19 56 57 58 59	61	0 1 24 25 32 33 40 41	97	16 17 32 33 40 41 48 49
26	24 25 26 27 32 33 34 35	62	0 1 24 25 32 33 48 49	98	16 17 32 33 40 41 56 57
27	24 25 26 27 40 41 42 43	63	0 1 24 25 32 33 56 57	99	16 17 32 33 48 49 56 57
28	24 25 26 27 48 49 50 51	64	0 1 24 25 40 41 48 49	100	16 17 40 41 48 49 56 57
29	24 25 26 27 56 57 58 59	65	0 1 24 25 40 41 56 57	101	24 25 32 33 40 41 48 49
30	32 33 34 35 40 41 42 43	66	0 1 24 25 48 49 56 57	102	24 25 32 33 40 41 56 57
31	32 33 34 35 48 49 50 51	67	0 1 32 33 40 41 48 49	103	24 25 32 33 48 49 56 57
32	32 33 34 35 56 57 58 59	68	0 1 32 33 40 41 56 57	104	24 25 40 41 48 49 56 57
33	40 41 42 43 48 49 50 51	69	0 1 32 33 48 49 56 57	105	32 33 40 41 48 49 56 57
34	40 41 42 43 56 57 58 59	70	0 1 40 41 48 49 56 57	106	0 8 16 24 32 40 48 56
35	48 49 50 51 56 57 58 59	71	8 9 16 17 24 25 32 33		

The last graph shows the bandwidth when memory is allocated interleaved. The labels on the `x` axis show the processors in which the threads were running: cores 0-7 are on P0 (node 0), cores 8-15 are on P1 (node 0), . . . , cores 56-63 are on P7 (node 6). The red dashed line is the average.

As one can see, the bandwidth can range from as low as 1.6 GB/s to as high as 4 GB/s, depending on where threads and data are allocated. Thus, in memory-bounded applications, the different bandwidths translate into different waiting times for memory

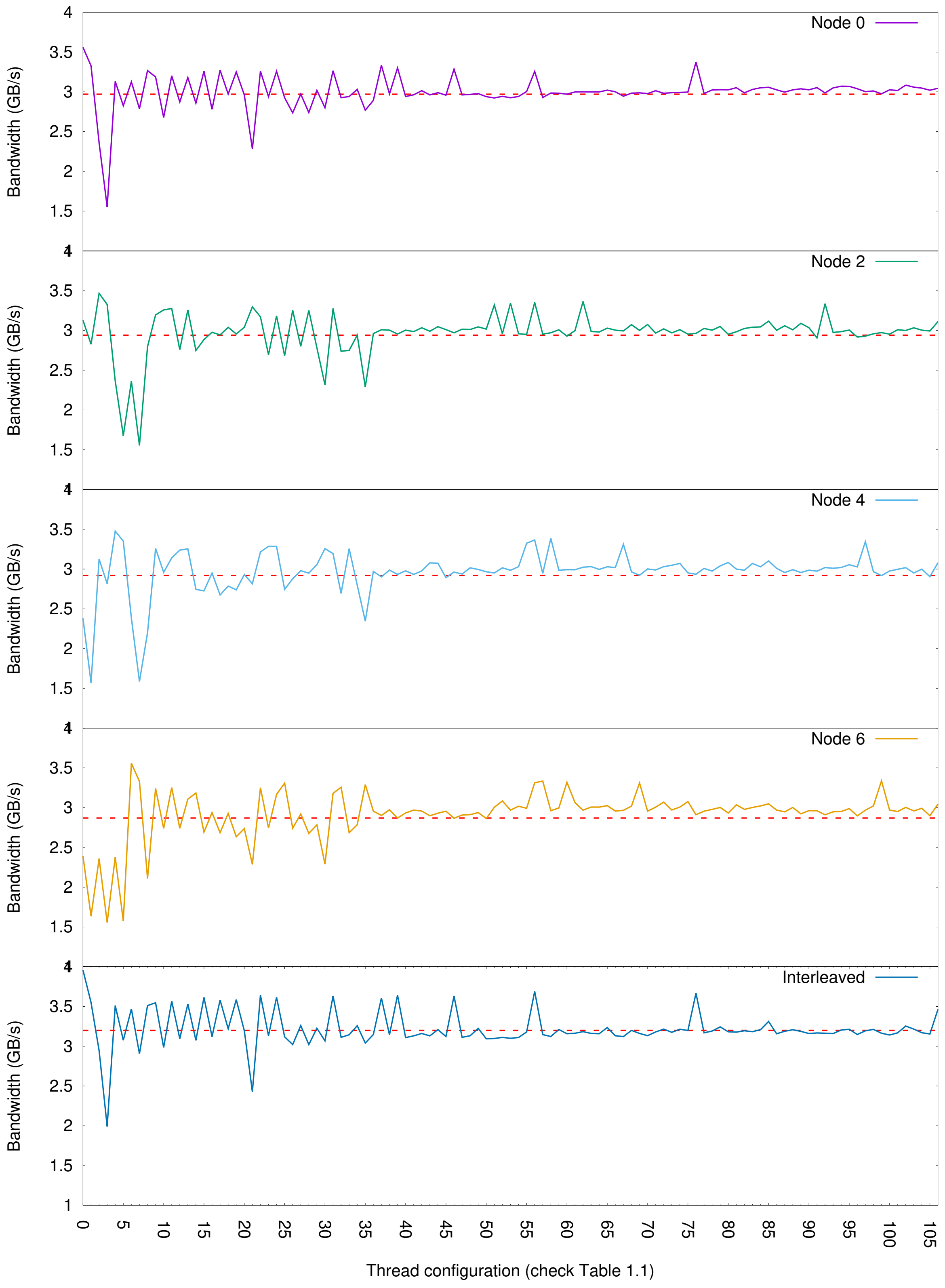


Figure 1.2: Bandwidth with different thread distributions

requests. Because the application is halted waiting for the memory request, its total running time is also impacted. Hence, this shows the importance of maximizing memory bandwidth usage.

An interesting observation from this experiment is that there is neither an universal optimal thread distribution nor an universal optimal data allocation: a thread distribution that works well for a particular data allocation may not necessarily yield high bandwidth with another data allocation. For example, for thread distribution `24-25-26-27-28-29-30-31`, the optimal bandwidth is obtained when all data is on node 2 (3.32 GB/s), whereas the worst bandwidth happens when data is on node 6 (1.56 GB/s). Node 2, however, is not always the optimal choice for data allocation – actually, it the worst choice for thread configuration `56-57-58-59-60-61-62-63` (1.55 GB/s). Thus, one can see that no particular memory or thread allocation is guaranteed to sustain high bandwidth on all configurations.

On average, interleaved allocation performs better than the other allocations (3.20 GB/s for interleaved vs. 2.87-2.97 GB/s for the other nodes). This is not difficult to explain, as the pressure on memory controllers and the traffic on interconnection links is distributed, increasing parallelism and decreasing congestion. While this is true for the average, it is easy to see that there is often a better thread allocation than interleaving. For instance, consider thread distribution `16-17-18-19-24-25-26-27`. For this distribution, interleaved allocation has bandwidth of 2.94 GB/s. Allocation on nodes 0, 2, 4 and 6, has, respectively, 2.35, 3.47, 3.12 and 2.36 GB/s. This happens because cores 16-19 are on node 2 and cores 24-27 are on node 4. In this case, even though allocating data exclusively either on node 2 or on node 4 increases contention, it also increases data locality. The latter, however, has a more significant impact on bandwidth than the former (in this particular case, at least), thus increasing bandwidth.

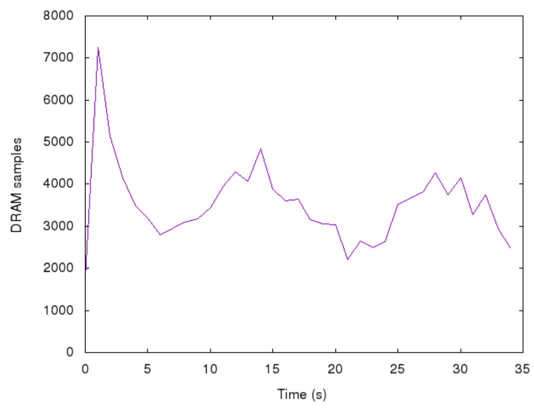
This simple experiment shows that correctly assigning threads and data is important for performance on NUMA machines.

1.3 A motivating example

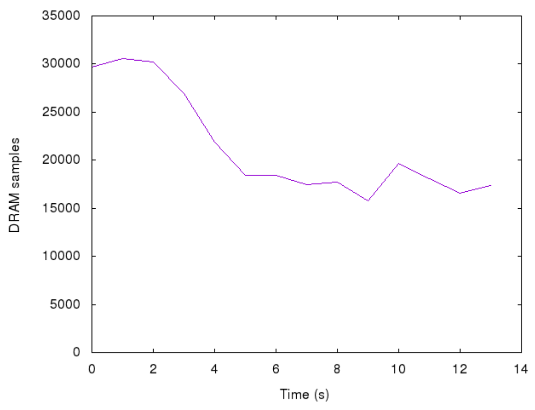
`Fluidanimate` is a Parsec benchmark [7] that simulates a fluid's motion over time, for animation purposes. It discretizes the fluid into 3-dimensional particles and interpolates intermediate results. Parallel threads simultaneously simulate multiple particles and each particle only interacts with neighboring cells. Therefore, a thread operates mostly within its set of private memory pages, with low sharing (at boundaries).

`Canneal`, another Parsec benchmark, minimizes the routing cost of chip design. It is a lock-free implementation of an application that tries to swap pseudo-random elements from two arrays. It runs for a fixed number of iterations (input-dependent), also with multiple parallel threads. The pseudo-random memory access pattern represents a worst-case scenario in terms of memory accesses, as caches and memory prefetchers are not very effective. Unlike `fluidanimate`, it is a much more regular program, in the sense that it has a single kernel that is repeated multiple times.

Figures 1.3a and 1.3b respectively show the memory profile of `fluidanimate` and `canneal`, as the number of memory accesses per second that miss all cache levels and are supplied by the main memory (DRAM samples). The different phases of `fluidanimate` can be clearly seen, as the number of samples changes considerably during program execution. The samples on `canneal`, on the other hand, follow a more constant pattern, result of its single phase.



(a) `fluidanimate`



(b) `canneal`

Figure 1.3: DRAM samples

These programs reflect very distinct behaviors and memory profiles. Both are, however, very sensitive to inefficient memory affinity choices. For example, both store most of their data in large `C++` vectors, which are initialized by a single thread. Thus, Linux's default first-touch policy allocates all memory on a single node. Remember that `canneal`'s memory access pattern is random, that is, all threads may access any position in the vector. Hence, a congestion effect occurs. Simply by changing allocation to an interleave pattern reduces running time by half. `Fluidanimate` also benefits from an interleaved allocation, although not as much. This happens because `fluidanimate` does not have as much sharing as `canneal`. For this benchmark, the optimal choice is to migrate all data to the node where the threads that access it are.

The performance degradation due to poor placement choices shows the importance of matching the program's memory access pattern to a correct allocation. Since it is not always easy to identify and/or manage what the pattern is (from the programmer's perspective), it is very rare to see programs that exploit NUMA machine's intrinsic memory layout.

PTB is a Linux kernel module that automatically monitors what memory pages are being used by a parallel program, cluster and migrate threads that have similar memory access patterns and migrates memory pages in order to increase data locality and decrease congestion on memory controllers. The rationale behind PTB is that bandwidth is limited and that an improper thread/memory distribution can create unnecessary bottlenecks on memory controllers and interconnect links. PTB produces a speedup of 1.45x for `fluidanimate` and of 1.84x for `canneal` over the default Linux scheduler.

Chapter 2

Related Work

Several techniques and models have been proposed in the literature to try to deal with the problem of page distribution in NUMA machines, many of which are available in current systems. In this section, we present some alternatives and comment their applicability.

2.1 Possible Approaches

2.1.1 Programmer control

The most straightforward way to control the memory layout in NUMA machines is to pass control to the programmer, through libraries [2]. Clearly, this technique has great flexibility and, if used by an experienced programmer that understands in detail the application's memory access pattern and the available hardware, can obtain significant performance gains. However, this practice demands expertise from the programmer and can lead to performance losses if used wrongly.

Library functions empower the programmer to set memory policies program-wide or to specific memory ranges [2]. These policies include: determining how memory should be allocated: on a particular node (`numa_alloc_onnode()`), on the current local node (`numa_alloc_local()`) or interleaved (`numa_alloc_interleaved()`); moving memory pages for specific nodes (`numa_move_pages()`); migrating all of a processor's memory to another node (`numa_migrate_pages()`); pinning the execution of a process to a set of nodes (`numa_run_on_node_mask()`) and setting from which nodes memory can be allocated (`numa_set_membind()`).

The combination of these functions gives the programmer great flexibility. However, not only does it impacts development and maintenance, it also compromises portability. Hence, they are recommended for developers who have great architectural knowledge and for applications designed to run on specific hardware.

2.1.2 Allocation Policies

An allocation policy is a set of guidelines passed to the operating system that determines how memory should be allocated. Modern operating systems support various policies [28, 5], including:

Local Allocate memory on the node that requested it.

Binded Allocate memory only in a predetermined set of nodes, regardless of which node requested it. If there is not enough memory in the specified nodes, return an error.

Preferred Allocate memory preferably in a set of nodes. If there is not enough memory in the specified nodes, try using other nodes.

Interleaved Allocate memory in round-robin fashion in a predetermined set of nodes.

First touch Allocate memory on the node that first touches (read or write) the data.

Next-touch Migrate data to the next node to touch the data.

Allocation policies remove from the programmer the responsibility for memory distribution and can give significant performance gains. Majo and Gross [34] exemplify this with an application in which a thread allocates and initialize all memory and then all other threads, running on multiple nodes, process on this data. This is a common design in parallel applications. A local policy would allocate all memory on only one node, while the interleaved policy would distribute it to all nodes. In the first case, a few threads would have only local accesses, while the rest would only have remote accesses. In the second case, all threads will have both remote and local access, generating a performance gain of 35%. On the other hand, a program in which each thread allocates its own memory and has no memory sharing would benefit from a local allocation policy.

The next-touch migration policy is useful when the application changes the memory access pattern [51]. Lof and Holmgren show that the next-touch policy, when used appropriately, can produce a speedup of up to 64% [31].

Allocation policies can be defined at various levels. There is a global policy (local allocation by default), policies that can be specified at the time of invocation of a program (through tools such as `numactl` [28]) or for specific ranges of memory, as explained in Section 2.1.1.

Even though allocation policies require less from the developer, its usefulness is still limited because it also requires knowledge of the program's memory access patterns. Furthermore such policies are static, meaning that changes in memory access patterns within the program can not be accommodated, except for direct programmer intervention. Finally, memory migration is not possible with these techniques.

2.1.3 Compilers

Compilers are very powerful tools and are capable of performing rich analysis that can be used to address the problem of memory locality.

Frameworks such as OpenMP provide a wealth of information that an application's runtime can use to make informed decisions on where to schedule threads [10]. In addition, the compiler can assist an application to discover where to allocate or migrate memory. Piccoli *et al.* have shown a technique like this that is able to accelerate the execution of a program by up to four times [42]. However, it is only applicable to a restrict set of applications.

Another field explored in compiler-assisted NUMA data placement is that of directive pragmas, hints passed on to the compiler by the programmer. These can and have been used to help the compiler distribute data more efficiently [22, 4]. Even though performance gains have been reported and its use is fairly simple, this approach presents problems similar to those shown in Section 2.1.1 because they require the programmer to possess a detailed knowledge of the available hardware and the application's memory access pattern. Furthermore, it compromises portability, as different platforms may perform better with different distributions.

On the other hand, if no directive pragmas are used, the compiler's job gets much more complicated. In this scenario, the parallelization goals and data locality may have different and conflicting requirements. While the first attempts to distribute data and computation between all nodes (for effective parallelization), the latter would favor having everything in one single node (for smaller latency) [39].

2.1.4 Programming Languages

To ease the burden of developing efficient parallel code, several programming languages have been developed in the last decades. These include `Charm++` [25] and `Cilk` [9] (and later `Cilk Plus`). In common, they provide abstractions to threads and cores and include automatic load balancing. However, they are not natively NUMA-aware, although some research has been done to include such support [43].

These languages are extensions to existing popular languages (such as `C++`) and they claim that it is easy to port existing code to their new languages. This claim only applies to code that was originally written in the popular programming language in which the new language was built on. Thus, this approach, even if they eventually become NUMA-aware, would be limited to a subdomain of programs and would require programmer intervention.

2.1.5 Operating System

In common to the techniques explained in Sections 2.1.1 and 2.1.2 is the fact that they are both unfeasible to complex large-scale applications because they require detailed knowledge of the application. A viable alternative approached by several authors to achieve an abstraction for this problem is to delegate distribution to the operating system.

To this extent, the operating system should be able to comprehend the memory usage pattern of each program. A reliable source for this information is the rate of each core's page-faults to each memory page [16]. With this data, the operating system can act to minimize the average latency to memory.

To accomplish lower memory access latency, operating system developers make use of two techniques or a combination of them, namely:

- **Page migration**: a page that was deemed ill-located can be migrated to another node. This technique involves several operations that introduce overhead: allocating a new page, updating pointers to the new physical address in the TLB, removing replicas and copying the old data to the new address [54].

- **Process migration**: instead of moving memory pages, one can move the process to a core closer to the data. This approach implies the loss of the "affinity" created between the core and the task because the cache content that had been filled is lost. However, Vaswani *et al.* show that the affinity loss is a smaller problem and can be compensated if done in a controlled manner [53]. Care must also be taken not to overload any core, while leaving others idle.

This approach's main advantage is its ability to adapt to different phases during a program execution. Moreover, it removes from the programmer and from the user any need to worry about underlying hardware characteristics.

There has been extensive work (both in industry and in academia) on Operating System-based automatic page migration. One of the earliest works was on the Stanford DASH, a cache-coherent multiprocessor [11]. Chandra *et al.* compare several scheduling policies [11]. Because their experimental platform required excessive locking to perform page migrations, they followed a trace-based approach to show the potential benefit of page migrations. Furthermore, they compare policies based on cache misses and TLB misses and show that they can be almost as effective.

Another early approach was featured in the SGI Origin 2000 [13, 24], whose IRIX operating system had an automatic page-migration mechanism. This machine had hardware counters that could keep track of the ratio between remote and local accesses. When a threshold is reached, an interrupt is generated to inform the operating system; the actual migration depends on several filters, including an algorithm to prevent a page from ping-ponging.

More recently, the Linux kernel community saw an intense debate on whether page and/or process' migration should be adopted [14]. Metrics and basic algorithms to dynamically understand a processes memory access patterns and to control their migration were introduced in the Linux kernel [16], including both thread and page migrations. Tests performed by the Linux community showed that this approach works very well in programs with fairly regular access patterns [21]. However, as will be shown in Chapter 3, experiments show that the automatic balancing produces performance loss of more than 60% under certain situations.

A more thorough explanation of how Linux's NUMA scheduler works and how it balances page and thread migrations is provided on Chapter 3.

2.2 OS-related work

PTB is an Operating System approach to the NUMA distribution problem. In this section, a literature review of previous contributions related to this approach is presented. Such work includes the collocation of threads that share data into the same node, the design of schedulers that are aware of congestion, the maximization of of asymmetric bandwidth usage, and the detection of memory access patterns.

2.2.1 Thread clustering

Kamali shows the benefit of collocating threads that share data in the same processor to take advantage of shared caches [26]. They use cache-related hardware events (snoops and cache misses) to estimate how much data sharing there is between two threads and whether there would be a benefit to migrating threads (considering contention effects). Their approach, however, can only cluster a small number of threads (equal to the number of nodes in the machine), because the events that they used are associated to chips and do not provide information per thread. PTB, on the other hand, as described in Section 5.1, uses a special set of hardware events that provides the addresses of memory operations and allows for a finer-grained control for programs executing with many threads.

Tam *et al.* also use hardware events to capture accessed memory addresses and they also use a thready similarity index similar to PTB, the difference is that their index is not normalized [49]. Non-normalized indexes may cause threads with similar access patterns, but with lower access counts, not to be co-located in the same node. By normalizing the index when comparing different threads, PTB favors memory access patterns rather than their absolute number of accesses. Furthermore, Tam *et al.* use a clustering algorithm that can split similar threads into different nodes. By using a partition algorithm, PTB guarantees that similar threads are co-located.

2.2.2 Congestion-aware scheduler

Majo *et al.* perform an evaluation of the impact of bandwidth sharing on multicore computers [34]. By measuring the bandwidth for different numbers and combinations of local/remote threads, they show that in parallel applications, optimizing for data locality not always yields best performance, as memory controllers bandwidth limitations are also important. Thus, they recommend that not only software developers, but also operating systems and compiler (runtime) developers investigate the balance between data locality and congestion alleviation.

Dashti *et al.* introduced a congestion-aware memory placement algorithm [17] that uses precise hardware events to track memory accesses. Their algorithm is composed of four mechanisms: thread clustering, page co-location, page replication and page interleaving. For thread clustering, they use Kamali's algorithm. The fate of each page is determined locally. The first option is co-location: this happens when a memory page is accessed by a single thread or by a set of threads co-located in the same node. By migrating the page to that node, they reduce remote accesses. If a page is accessed by threads in different nodes, they evaluate if the page is a good candidate for replication, which happens only if it is rarely written to. Unfortunately, the synchronization overhead required by each write would be too big. Replication, when done correctly, decreases remote accesses. Finally, the page is marked for interleaving. Their interleaving approach favors migrating pages to lightly loaded nodes. Unlike PTB, their solution does not consider the asymmetry in NUMA architectures. As detailed in Section 5.1, PTB uses a modified version of their profiling tool in its application profiling phase.

2.2.3 Bandwidth Asymmetry

Lepers *et al.* show that not only are NUMA machines non-uniform, but they are also asymmetric [29]. They show the benefit of scheduling threads and placing memory in a way that maximizes bandwidth usage between nodes. Their solution tackles the problem of applications running with less threads than there are cores available. It tries to predict which subset of nodes would be optimal for the program. PTB, on the other hand, is an approximated solution for the harder problem of scheduling an application that uses as an arbitrary number of threads.

2.2.4 Memory access pattern

Detecting the memory access pattern is fundamental to any algorithm that attempts page and/or thread migration. Common solutions include a modified page table [18, 16] or hardware events [17, 29]. Modified page tables require a special kernel but they are independent of the hardware while hardware events are specific to each hardware but do not require a custom kernel. PTB uses a hardware-event-based approach.

Modified page tables work by intercepting page faults. At every time step, the “present” bit is cleared from the page table. When a page fault occurs, the modified kernel intercepts it and stores the ID of the thread that generated it. This is not a true page fault, and the addresses translation can be quickly returned to the application, keeping the overhead low.

The hardware-based approach used to detect memory access patterns uses modern hardware events, such as Intel PEBS[23] and AMD IBS [3]. Traditional hardware events work only as counters and do not provide any more information about who or what caused the event. Modern events, on the other hand, are more precise, in the sense that they provide more information about the event. Of relevance to this work, the information of which memory address was accessed and which core generated it (instead of which *node*). This enables developers to build a detailed memory access map. Dashti *et al.* [17] created and released a profiling tool that uses IBS to create a map of threads’ accesses to pages, which they use in *Carrefour*, their congestion-aware scheduler. The profiling mechanism used in PTB is a variation of that tool. Section 5.1 explains how PTB uses the hardware-based approach.

Chapter 3

Linux's NUMA Scheduler

Until version 3.8, Linux's support for NUMA was deficient [15]. There were competing solutions under development and there was little code integrated in the Linux development tree [14]. On that version, the foundations for automatic NUMA balancing were introduced by Mel Gorman [16], including the first policy to be merged in the main development tree: *Migrate on Reference Of pte_numa Node* (MORON), which the author himself described as "a very stupid greedy policy", but also noted that "it can be faster than the vanilla kernel and the expectation is that any clever policy should be able to beat MORON".

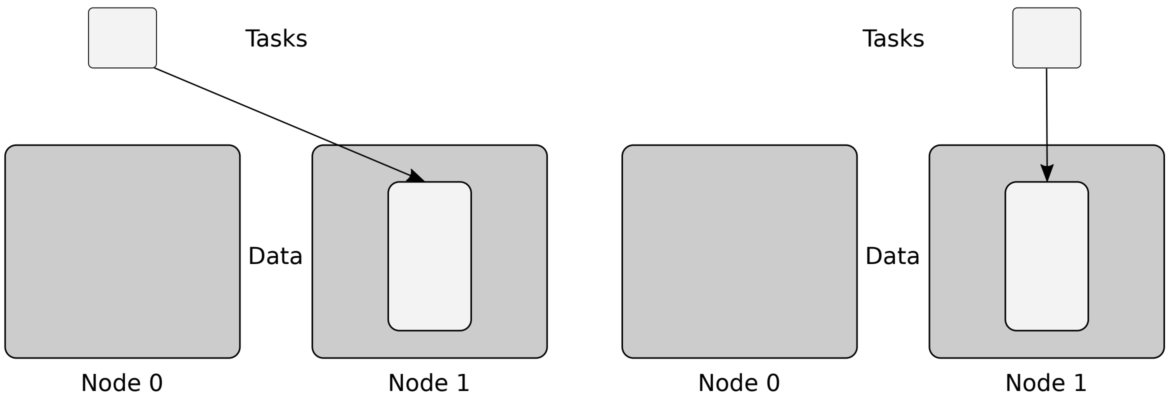
On version 3.13, further support and better algorithms were merged into the tree, including an algorithm that tracks which pages a process is accessing and groups threads that share data into "NUMA groups" [16]. Page tracking is done with the modified page table described in Section 2.2.4 ("false page-faults"). Because page migrations are expensive, a page is only migrated if it is accessed twice from the same NUMA node or by the same task. Grouping threads that share data involve a complex set of rules, which requires tracking the last thread to access each page. When two threads access the same page, they are put into the same group; the scheduler will favor co-locating threads in the same group.

The kernel developers were aware that the algorithm should be able to adapt to different situations. For that, they included a set of variables, configurable by the machine administrator, that control how often and how much memory is invalidated in each cycle [1]. Furthermore, the kernel is able to automatically adapt its invalidation frequency, given how many local and remote faults happened in the last cycle. If there were many local accesses, then probably the balancing is already correct and thus the interval between each cycle is increased. On the other hand, if remote accesses dominate, then this is an indication that there is still a considerable imbalance, and therefore the interval is decreased, in order to accelerate pages/processes migration.

The kernel is flexible and can apply either a CPU-follows-memory or a memory-follows-CPU policy, or a mixture of both, as the next examples show.

Thread migration

The simplest scenario happens when threads and data are in different nodes and both can be migrated (migration limitations are discussed in the next examples). To test which strategy the kernel favors, the following simple experiment was performed: allocate a large array on a remote node and access it with a single thread until either the thread is migrated or the pages are migrated, as Figure 3.1 shows. The experiment showed that, given the choice, the NUMA scheduler will migrate the thread. This is the obvious and correct choice, as page migrations is much more expensive than thread migration [33].



(a) Before (b) After

Figure 3.1: Thread migration

Page migration

However, not always is thread migration desirable. A simple scenario in which this is true is if there are more threads sharing data than there are cores in a node, because this would cause threads to compete for CPU time, hindering the application.

In this case, page migration may help to improve performance. For simplicity, suppose there are 2 NUMA nodes, each with the same number of cores, as Figure 3.2 shows. All cores in both nodes are occupied by threads that share data and there are not enough available cores in neither node to receive all threads. Hence, thread migrating is not desirable. Additionally, there is an imbalance between the number of pages in each node, as Figure 3.2a shows. The scheduler migrates pages from the more heavily used node (Node 0) to the less heavily used node (Node 1) to try to balance them and consequently increase bandwidth availability, as Figure 3.2b shows.

Thread and Page migration

Finally, a case in which both threads and pages are migrated is shown in Figure 3.3. In this scenario, there are two NUMA groups, that is, two sets of threads that share data inside the set but not between sets (threads and data from the same group have the same color in the figure). There are threads and pages from both NUMA groups distributed between the two nodes (Figure 3.3a). The algorithm moves all threads and pages of each NUMA group to a single node, increasing locality (Figure 3.3b). Simply put, the

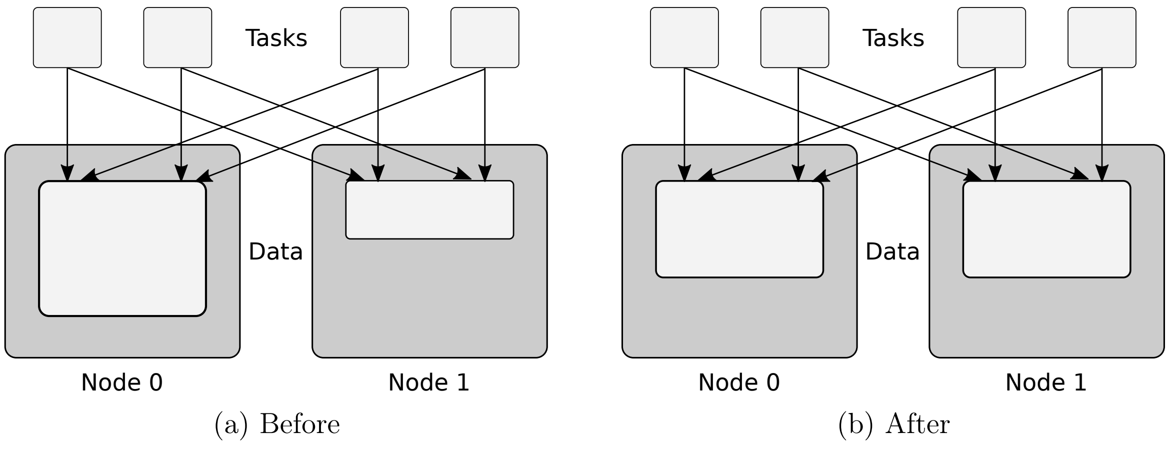


Figure 3.2: Page migration (adapted from [52])

algorithm works by comparing the ratio of local/remote accesses of each thread T in the current node (C) to a desired target node (D). If T has more accesses to D than to C, the algorithm tries to find a thread T' from D that would benefit from moving to C (or whose downside would be smaller than moving T to N). This process is repeated until there are no more potentially beneficial swaps.

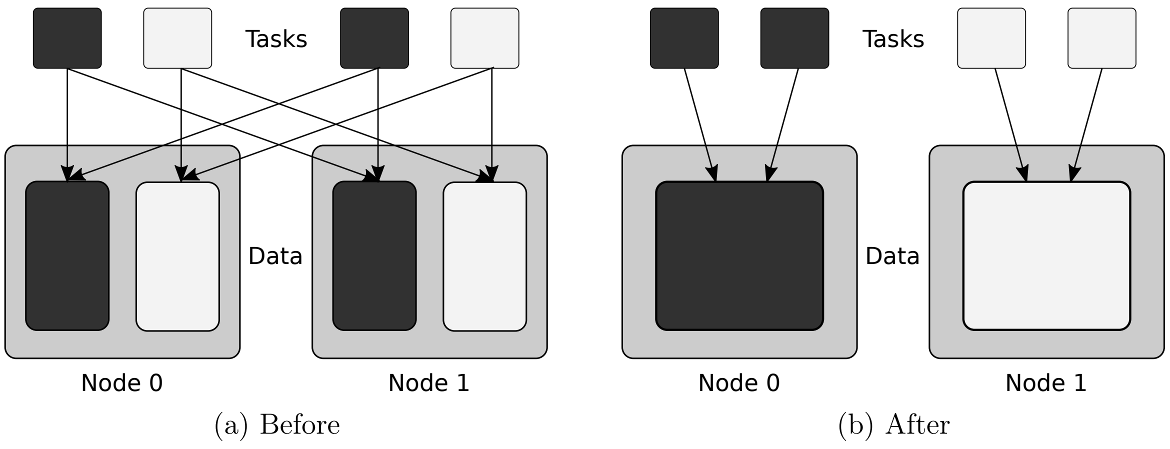


Figure 3.3: Task grouping (adapted from [52])

Issues with Linux's NUMA scheduler

According to the algorithm's main developer "some reports indicate that the performance is getting close to manual bindings for some workloads but your mileage will vary". However, it was found that for several applications, not only did Linux's NUMA automatic balancing failed to speed applications up, but it actually decreased performance. For instance, applications `is` and `cg` from the NPB benchmark suite had a speedup of 0.6x and 0.75x, respectively. Others had a marginal speedup, far from the possible performance of manual binding.

These results alone mean that the problem was far from solved. Additionally, the algorithm is agnostic to heterogeneous NUMA topologies [52], an aspect present in the machine used in this work.

Chapter 4

Modeling NUMA Architecture Performance

Given its large configuration space, NUMA architectures are inherently complex [46]. Thus, understanding a NUMA architecture memory and interconnect model is not trivial. Furthermore, architectural designs are in constant evolution and their details are not always publicly available. In addition, intercommunication links can have asymmetric bandwidths mainly due to three causes. First, nodes usually do not form a fully connected graph, thus the number of hops required to reach a remote node can vary. Second, interconnect links' width and bandwidth may be different [3]. Finally, the link's bandwidth might be different for both directions [29]. As a result, bandwidth information available in the machine's manual is often not enough to build a precise model of the intra and inter node's communication.

Experimental platform

Consider, for example, Figure 1.1 that shows the topology of the machine used in this work: an AMD Interlagos 6272 computer containing 4 NUMA nodes. Each node has 2 processors and each processor has 8 cores. At each node, one processor is located in the *upper plane* (P1, P3, P5 and P7) while the other is in the *lower plane* (P0, P2, P4 and P6). Processors in the same node (*e*.g. P0 and P1 @ Node 0) are connected with a 16-bit link, while all other connections are 8-bit wide. In addition, memory distribution across the machine's nodes is asymmetric, *i.e.* there are only 4 memory banks (16 GB each), one attached to each lower-plane processor (P0, P2, P4 and P6). Therefore, cores from different processors of the same node (*e.g.* P2 and P3 @ Node 2) might have different memory latencies and bandwidths. For the sake of simplicity, memory attached to one of the lower-plane processors (P0, P2, P4, P6) is regarded as memory attached to the nodes containing the corresponding processor (N0, N2, N4 and N6, respectively), without specifying to which processor it is directly connected to.

4.1 Performance characterization

PTB uses a simple profile-based characterization procedure to capture the performance of the underlying hardware into an architecture model. To do so, intra-node and inter-node bandwidths of the machine are experimentally and automatically determined once through a per-thread STREAM-like program [37], that performs pseudo-random memory accesses to estimate local and remote node bandwidth. Pseudo-random memory accesses are used to avoid prefetching mechanisms and their impact in the measured bandwidth. However, traditional pseudo-random generation functions might impose important overheads on the execution of the profiling. Thus, a simpler method based on the remainder of prime-number multiplication is used. The use of an appropriate prime numbers minimizes resonance [40] and, as far as the prefetcher is concerned, memory accesses generated by this method are very similar to random accesses. An algorithmic view of the method is shown in Algorithm 1.

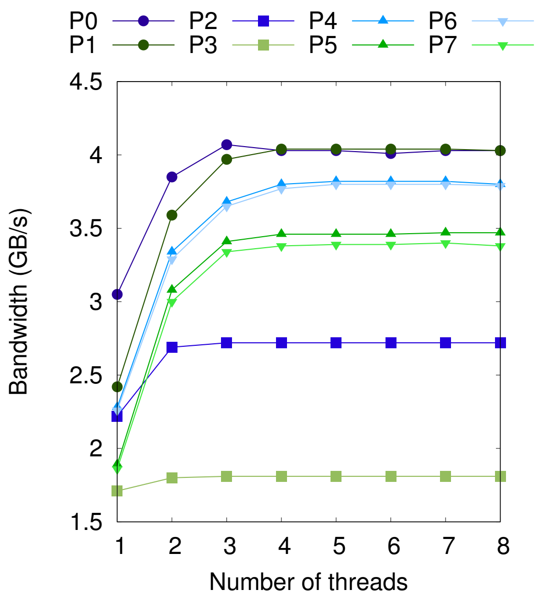
Algorithm 1: Pseud-random memory accesses

```
void kernel()
{
    #pragma omp parallel
    {
        t = omp_get_thread_num() + 1;
        prime = 302400041;
        for (i = 0; i < LENGTH; i++) {
            p = ((i + LENGTH/NTHREADS * t) * prime) % LENGTH;
            A[p] = B[p] + scalar * C[p];
        }
    }
}
```

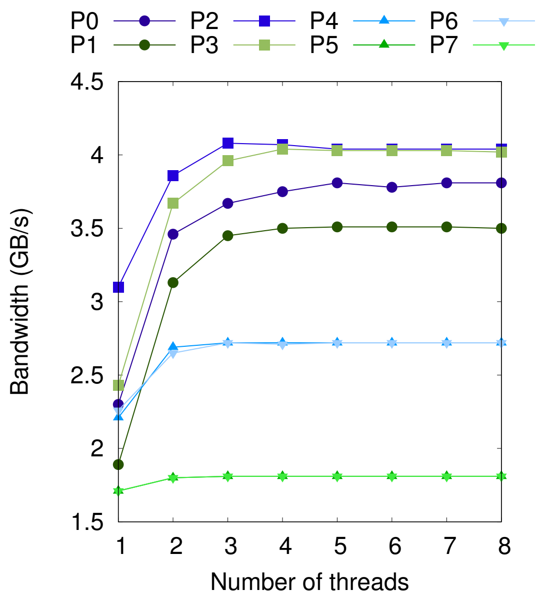
The empirical characterization of the machine's performance requires two measurements. The first experiment estimates the memory bandwidth between processors of all nodes when only one processor is running the characterization experiment. In the second experiment, all nodes are running the experiment. The difference between these experiments is that while the first captures the processor's maximum bandwidth without interference from other processors, the second experiment measures the bandwidth in a stressed scenario, with all processors competing for resources. It is important to remember that it is not enough to profile each one of the 4 nodes because processors in the same node have different inter-node connections and thus routing and/or number of hops are different.

Bandwidth with one active processor

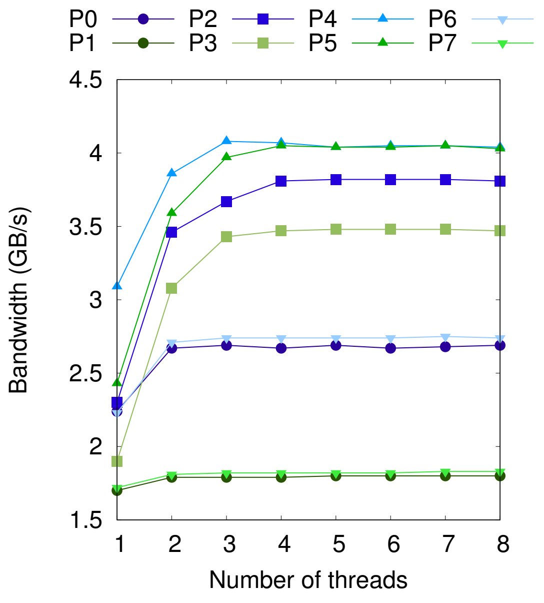
Algorithm 1 was executed with a different number of threads (always in the same processor) and the results are shown in Figure 4.1. Data is allocated exclusively in one node, as identified in the figures' caption. The curves in the figure corresponding to the lower-plane processors are in blue and those from the upper-plane processors are in green. Also, data points corresponding to processors belonging to the same NUMA node are represented by the same symbol.



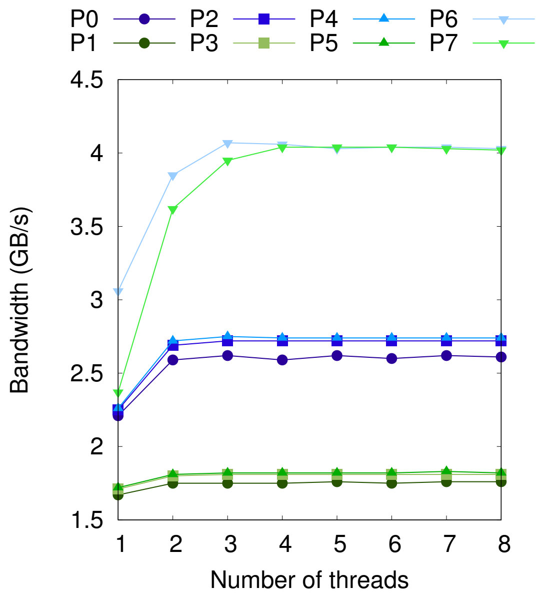
(a) Allocated on node 0



(b) Allocated on node 2



(c) Allocated on node 4



(d) Allocated on node 6

Figure 4.1: Bandwidth between all processors and nodes

For all nodes and processors, the highest bandwidth is achieved with 3-4 threads, after which there is a plateau. Local accesses achieve the maximum specified bandwidth, roughly 4 GB/s, independent of the plane where the processor is located. But processors in the lower plane have slightly higher bandwidth for a lower number of threads. For instance, in Figure 4.1a the curve for P0 (lower plane) raises faster than the curve for P1 (upper plane). For some remote nodes, the sustained bandwidth of the processor in the lower plane is higher than that of the processor in upper plane. For example, in Figure 4.1c, Processor P6 (lower plane @ N6) has a higher sustained bandwidth than processor P7 (upper plane @ N6). Both results are expected because a processor in the lower plane require one less hop to access any memory bank the than a processor on the upper plane.

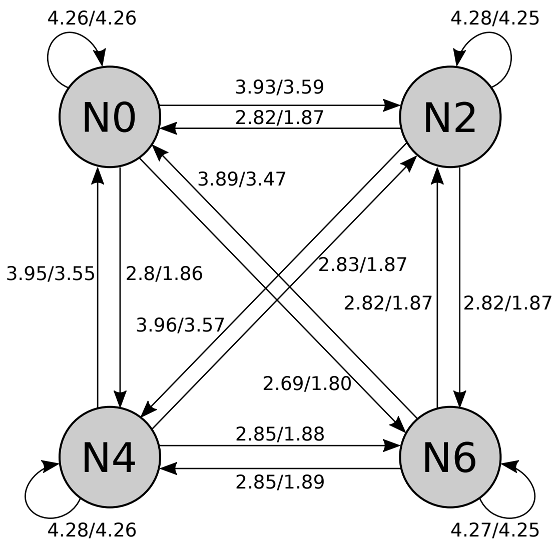
Some interesting insights on the performance of the machine are revealed when comparing the multiple graphs of Figure 4.1. By examining the topology of the machine shown in Figure 1.1 it would appear that the bandwidth to access memory allocated in Node 0 from a thread running in processor P2 should be similar to the bandwidth to access memory in Node 2 from a thread running in processor P0. However, the experimental data in Figure 4.1 shows that this is not the case. P2 bandwidth to Node 0 is 2.82 GB/s while P0 bandwidth to Node 2 is 3.93 GB/s. This unexpected asymmetric bandwidth can be due to a number of factors, including routing algorithms, difference in link speeds for each direction, software delays, queue contention, among others. The advantage of the above characterization approach is that the end-to-end characterization will capture all such complex effects in a single number. The downside is that the characterization has to be repeated whenever the hardware operation changes, e.g. after adding new memory banks, although this is an infrequent event. PTB uses the above described architectural model to guide its bandwidth allocation heuristic.

4.2 The Bandwidth Graph

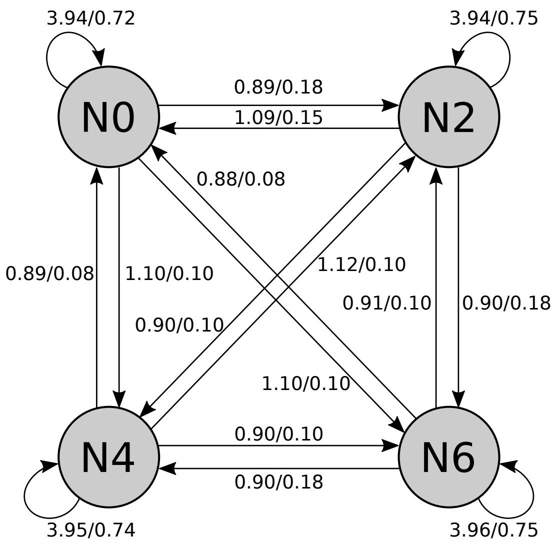
The machine's maximum bandwidths can be summarized by means of the Bandwidth Graph (BG) shown in Figure 4.2a. The directed edges in this graph flow from the source of a memory access to its destination. Two numbers are associated to each directed edge: the first is the lower plane's processor bandwidth and the second is the upper plane's processor bandwidth. For example, in edge N0 $\rightarrow$ N4 the maximum bandwidth available from accesses in a thread running at P0@N0 to the memory in N4 is 2.80 GB/s while from P1@N0 to memory in N4 the maximum bandwidth is 1.86 GB/s. Finally, it is important to observe that this graph shows all connections between processors and memory, even though a physical connection might not exist. For example, there is no physical connection from P7@N6 to the memory in N0, but the characterization procedure estimates a 3.47 GB/s bandwidth. Since they are not directly linked to memory, accesses must make at least two hops to reach any memory bank, thus impacting performance, which is captured in the BG.

Bandwidth with multiple active processors

For the experiments reported in Figure 4.1, only one of the eight processors was active. This is an optimistic scenario. Programs use multiple processors and often multiple programs are running simultaneously and competing for memory bandwidth. The second experiment measures bandwidth under such conditions. This experiment also allocates all memory in only one node, but now all cores from all processors execute the memory-intensive code of Algorithm 1, stressing the memory system. The result of this experiment is shown in Figure 4.2b.



(a) Bandwidth graph



(b) Bandwidth graph with full load

Figure 4.2: Bandwidth graph

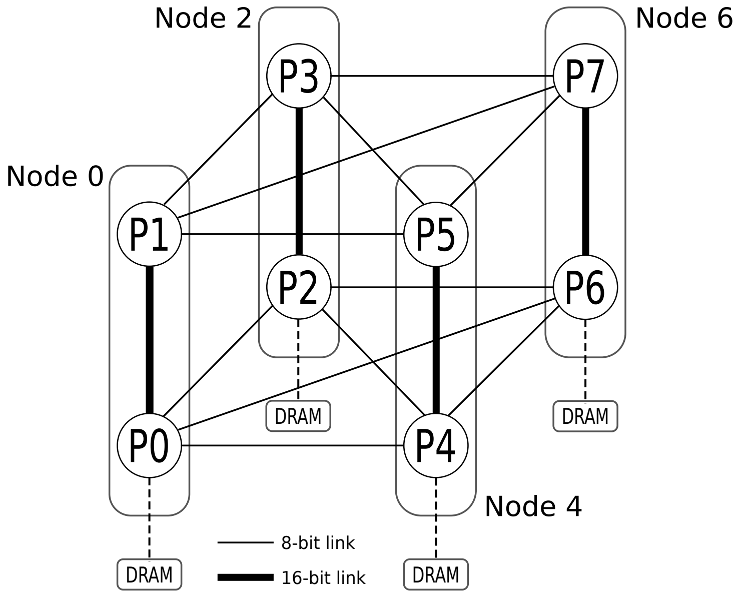


Figure 4.3: Experimental platform (adapted from [12])[1]

[1]Figure 4.3 (same as Figure 1.1) is repeated for reader's convenience.

Similar to the previous experiment, asymmetry is present. Also, as expected, the measured bandwidth results were lower than those of the previous experiment. While the results reported in Figure 4.2a are the bandwidths' upper bound, the results from Figure 4.2b can be seen as their lower bound, *i.e.*, the bandwidth that can result when the memory controller and interconnect links are being stressed.

The Bandwidth Graph is a simple, but not simplistic model: it captures several features into one number. It is obtained through a mix of loads and stores and it incorporates routing and memory access overheads, TLB misses, local/remote latencies, etc. The Bandwidth Graph aims at being a generic and experimentally-obtainable bandwidth representation of any machine. Hence, incorporating other numbers into the model would make it too complex and defeat its generality purpose.

The main goal of PTB is to maximize bandwidth utilization while minimizing contention and memory access latency. Running the machine with bandwidth utilization close to those in Figure 4.2b could be very harmful to any memory-bound program. During page/thread allocation, PTB seeks to reach the maximum bandwidth from Figure 4.2a while minimizing congestion from that point on. To that end, it uses the results from the optimistic measured upper-bound bandwidth to guide the allocation of pages and threads across NUMA nodes. Using the pessimistic lower-bound estimation could preclude PTB from adequately use all the potential available bandwidth, as it could diagnose congestion before it actually happened or when it did not happen at all.

4.3 Congestion/remote access tradeoff

If memory bandwidth to a given node reaches peak performance with accesses from a few threads and the machine has several memory channels, it may be interesting to distribute memory pages across all nodes, even if it means paying the cost of a remote access [32]. The following experiment assesses this tradeoff. Assign 8 threads to a single processor (the number of cores in each processor), allocate all memory in that processor's home node and measure the application's memory bandwidth (similar to the previous experiments). Then, repeat the experiment but now with *part* of the memory allocated interleaved in the 3 other nodes (which initially had no other memory allocated). Repeat this process until all memory is allocated exclusively on remote nodes.

Figure 4.4 shows that the bandwidth *increases* as memory pages are moved away from the home node. Trading higher remote memory access latency for lower congestion in the memory controller pays off until around 60-70% of memory is remote. From this point on, the remote latency cost starts to be the dominant factor on the overall bandwidth. The different bandwidth between lower-plane and upper-plane processors shown in the graph can be accredited to their physical distribution, which requires from lower-plane processors one less hop to reach memory than from upper-plane processors. Thus, they enjoy higher bandwidth.

As Chapter 5 will show, PTB uses this knowledge to guide its data distribution in two manners. First, it uses a bandwidth estimation to distribute memory pages, favoring congestion alleviation instead of data locality (however, if two nodes are similarly loaded,

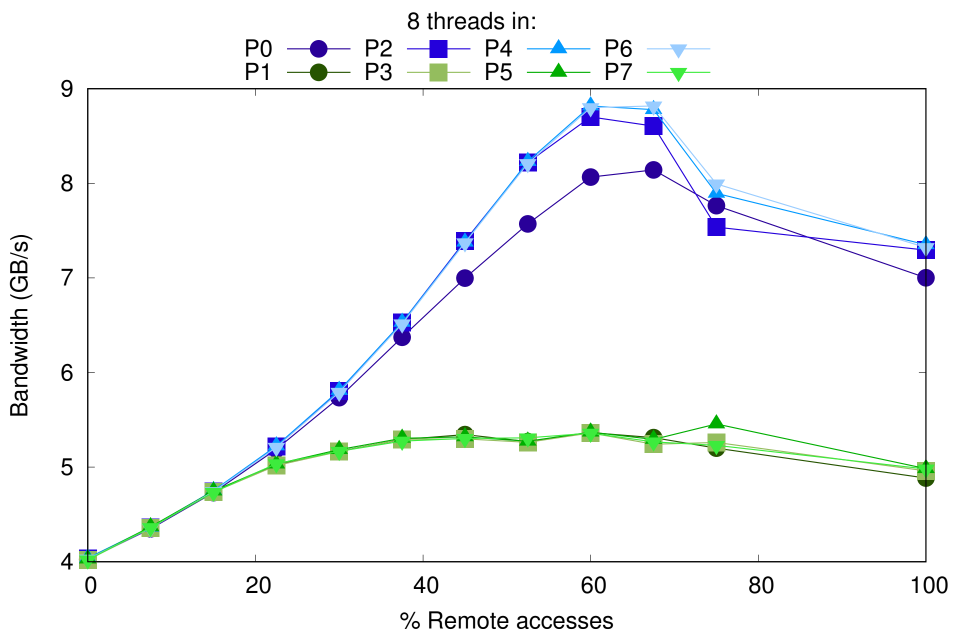


Figure 4.4: Congestion-locality tradeoff

the node with the highest number of accesses to the page will receive the page, to increase locality). Second, when PTB guesses that the memory subsystem is saturated, it uses interleaved allocation to equally distribute the load between all nodes.

Chapter 5

The PTB Algorithm

PTB is built as a Linux kernel module. Since modules can be added and removed from a running system, PTB does not require the user to compile a custom kernel.

PTB works in cycles and each cycle is composed of three phases: application profiling, thread migration, and page distribution. The duration of the profiling phase is an algorithm parameter. For the experimental evaluation this parameter was arbitrarily set to one second. Further studies of PTB may explore this parameter and determine if it is program dependent. During the profiling phase, the number of accesses per thread to each one of the pages is recorded in a *memory histogram*, where each bucket corresponds to a memory page, as Figure 5.1 illustrates. Based on this information, threads are migrated to available nodes and then memory pages are distributed to the nodes. A general overview of PTB can be seen in Algorithm 2 and detailed descriptions of each phase are presented in the following sections.

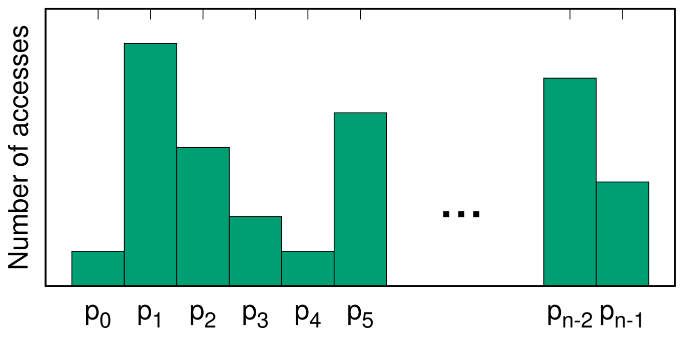


Figure 5.1: A memory histogram

Algorithm 2: Overview of PTB

```
1 while True do
2     Run profiler for specified time to build memory histogram
3     Distribute threads
4     Distribute pages
5 end
```

Notation and Definitions

Throughout this and the next sections and in the algorithms that describe PTB, several symbols and definitions are used. They are described below and summarized in Table 5.1.

$\boldsymbol{t}$ **and** $\boldsymbol{T}$ a thread and a set of threads;

$\boldsymbol{p}$ **and** $\boldsymbol{P}$ a page and a set of pages;

$\boldsymbol{n}$ **and** $\boldsymbol{N}$ a node and a set of nodes;

$\boldsymbol{A_t[p]}$ the total number of accesses of thread t to page p. It is bin p of thread's t memory histogram.

$\boldsymbol{b_n[p]}$ the estimated bandwidth from all threads running in node n to page p in the current PTB cycle. It is formally defined in Algorithm 6;

$\boldsymbol{B_n[n']}$ available bandwidth from node n to node n'. As PTB distributes threads and pages between nodes, it "consumes" bandwidth. Initially, it is set to the values from the Bandwidth Graph and is decremented by $b_{n'}[p]$ as p is allocated in n;

$\boldsymbol{\mathcal{B}_n[p]}$ cost function that evaluates how appropriate would be to allocate page p in node n. It is called `NodeScore` and is formally defined in Algorithm 7;

$\boldsymbol{M}$ a similarity matrix between threads. PTB defines a metric to compare how much the memory accesses of a pair of threads resembles. Similar threads are clustered and migrated to the same node;

$\boldsymbol{s = \{t | t \in T\}}$ **and** $\boldsymbol{S}$ A cluster of threads and a set of cluster of threads;

$\boldsymbol{D_t = \{(t, n) | t \in T, n \in N\}}$ as PTB decides that thread t should stay in node n, this tuple is added to this set. After all threads have been assigned a node, PTB iterates through this set and migrates all threads.

$\boldsymbol{D_p = \{(p, n) | p \in P, n \in N\}}$ Similar to D_t, but for pages.

$\mathtt{C}_1, \mathtt{C}_2$ are multiplicative constants used in the similarity index calculation and `NodeScore` function (Algorithm 7), respectively. Both have the objective of dealing with sampling's intrinsic inaccuracy and to avoid migrating due to small imbalances. The larger the value, the less likely is a thread/page to be migrated, as the difference between the current configuration and sampled configuration must be higher to force migration.

`Min_Acc` to prevent pages with few accesses from being migrated, it is required that it has a minimum number of hits. If it does not reach this threshold, it is ignored;

$\boldsymbol{\tau}$ controls how long sampling takes place until an interrupt is generated to reorganize threads and pages. It is the profiling cycle duration;

`ls` the cache line size;

Table 5.1: Notation

Symbol	Meaning
t	Thread
p	Page
n	Node
T, P, N	Set of {threads, pages, nodes}
$A_t[p]$	Number of accesses by thread t to page p
$b_n[p]$	Bandwidth from node n to page p
$B_n[n']$	Available bandwidth from node n to node n'
$\mathcal{B}_n[p]$	Cost function for page p and node n
M	Similarity matrix (between threads)
s, S	Thread cluster and set of thread clusters
D_t	Distribution of threads over nodes
D_p	Distribution of pages over nodes
$\mathtt{C}_1$,$\mathtt{C}_2$	Correctional factors (threads and pages, respectively)
`Min_Acc`	Threshold for considering a page
τ	PTB profiling cycle duration
`ls`	Cache-line size

5.1 Profiling

AMD Instruction Based Sampling (IBS) [19] is a profiling technique for AMD machines that provides detailed information about hardware events during program execution[1]. In particular, IBS provides the memory address accessed by a tagged instruction. As explained in Section 2.2.4, Dashti *et al.* developed an IBS-based tool to monitor page accesses, which is used in *Carrefour*, a congestion-aware scheduler [17]. PTB uses a modified version of that tool.

The thread distribution algorithm in PTB requires a histogram of memory-page accesses per *thread*, and the page distribution algorithm in PTB requires one histogram for each *node* (for the newly assigned thread affinities). Thus, PTB needs a fast way to aggregate threads histograms into node histograms. In *Carrefour*, the profiler used a Red-Black tree (per core) to record memory accesses that worked very well for their approach because they did not need to aggregate results. However, using this data structure with PTB required several traversals of the RB trees each time a histogram of accesses per node was needed because the algorithm had to find each page in every RB tree. An alternative solution would have been the use of a global RB tree. However, this solution would required locks, which made profiling too expensive.

To address this problem, the profiler was modified to use a hash table instead. The hash table is organized as follows and is shown in Figure 5.2. Every slot in the hash table corresponds to a memory page and contains a 64-position (parameterizable) array, one position for each core (and running thread). Slots are indexed by the page's base address and each slot keeps track of its corresponding address. Every time a memory access is recorded, PTB updates the slot's address of the corresponding page, without erasing the

[1]Intel-based machines have an equivalent feature called *Precise Event Based Sampling.*

corresponding array entry – thus, if two pages get mapped to the same bucket (collision), they will both increase the number of accesses in the same array, and it will seem that only one page (the last to record an event) was responsible for all the accesses. Slots are cache-aligned and updated without any synchronization mechanism. Obtaining the node memory histograms is then a simple matter of summing the correct indexes from the hash table.

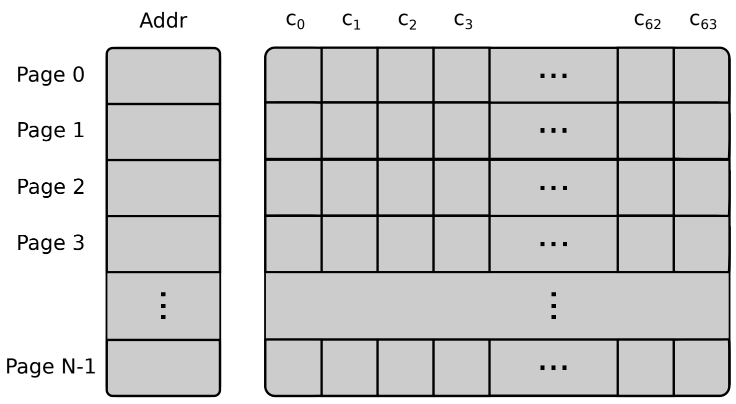


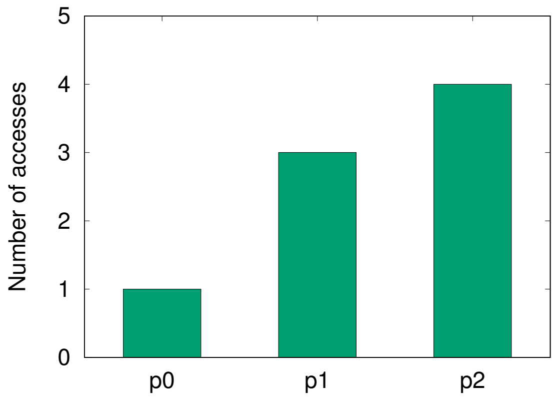
Figure 5.2: Profiling hash table

A synchronization-free data structure with a very simple hash collision strategy trades accuracy for performance: the hash table has constant-time insertion, update and lookup. Concurrent unsynchronized accesses to the hash table means that some updates might be lost. However, these losses do not change the output of the algorithm considerably because the sampling mechanisms employed are already inherently imprecise, and PTB does not need an exact representation of the memory access pattern. Thus the use of synchronization-free updates reduced the overhead of PTB considerably by reducing the profiling and thread/page distribution recalculation execution times.

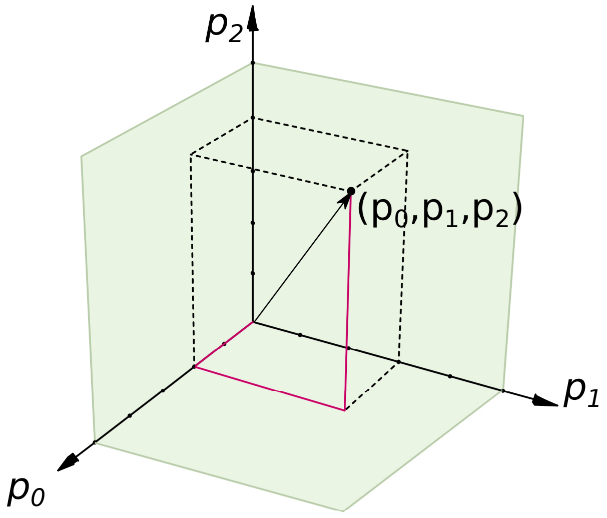
5.2 Thread distribution

The placement of threads that operate on the same memory pages onto the same processor has at least two advantages. The first is that one can move all the relevant pages to the node in which these threads are executing. However, a simple compact thread distribution strategy can be limited by congestion (*cf.* Section 5.3). The second advantage is that, because these threads are executing on the same processor, expensive inter-node cache line invalidations can be avoided therefore improving the efficiency of the caches.

To select which threads should run in the same processor, information gathered in the profiling step is used to calculate a similarity index between all pairs of threads. A thread's memory histograms can be thought of as multi-dimensional vectors, in which each dimension is represented by the number of accesses to a particular page. For example, consider Figure 5.3, in which a simplified 3-page histogram is shown in Figure 5.3a and the equivalent vector representation is displayed in Figure 5.3b. Two threads that access the same pages with similar frequencies have similar orientations (in the vector sense) and thus a high similarity.



(a) Histogram representation



(b) Vector representation (adapted from [47])

Figure 5.3: Memory accesses: equivalent representations

The cosine similarity [50] captures that property. It judges how similar the orientation of two vectors is and is very used in the field of data clustering [48] and information retrieval [38].

The cosine similarity between any two threads t_1 and t_2 is given by Equation 5.1.

$$\text{CosineSim}(t_1, t_2) = \frac{\sum_{p \in P} A_{t_1}[p] \cdot A_{t_2}[p]}{\sqrt{\sum_{p \in P} (A_{t_1}[p])^2} \cdot \sqrt{\sum_{p \in P} (A_{t_2}[p])^2}} \tag{5.1}$$

The cosine similarity falls in the range $[0, 1]$. Values close to 0 indicate low similarity between the threads' page access patterns, whereas values close to 1 indicate a high similarity. The rationale behind this approach is that a term in the numerator summation will only be high if the same page is highly used by both threads.Thus, a high cosine similarity indicates a similar angle between two vectors, regardless of the vectors' magnitude.

Sampling, however, is, by nature, inaccurate. This means that even if the memory access pattern is exactly the same in two PTB sampling steps, the memory histogram could not be; consequently, neither will be the Cosine Similarity. To avoid creating an imbalance that unnecessarily migrates threads, the Cosine Similarity is multiplied by a correctional factor $\mathtt{C}_1$ if two threads are already in the same node, which promotes conserving the current thread distribution (a description on how this and other parameters were chosen is given in Section 6.3). Hence, the similarity index between a pair of threads t_1 and t_2 is given by Equation 5.2.

$$M_{t_1t_2} = \begin{cases} \mathtt{C}_1 \cdot \text{CosineSim}(t_1, t_2) & \text{if } t_1, t_2 \text{ are in the same node} \\ \text{CosineSim}(t_1, t_2) & \text{otherwise} \end{cases} \tag{5.2}$$

The next step is to determine which threads should be co-located in the same node. PTB strives to prioritize the placement of threads with high similarities close to each other. To that end, it uses a graph-based approach. Let $G = (V, E)$ be a full graph where V is the set of all threads, and each edge in E is weighted by the similarity index of the threads it connects. Let G be the *similarity graph.* Consequently, thread assignment can be reduced to a minimum-cut graph partitioning problem, *i.e.* find a partitioning of the graph in which the sum of the weights of the edges between crossing partitions is minimal.

Unfortunately, the minimum-cut partitioning problem is NP-complete for an arbitrary number of cuts greater than two and is prohibitively expensive even for small fixed number of cuts [20]. The Kernighan-Lin algorithm [27], however, is an efficient heuristic when one wishes to partition the set in only two subsets. This algorithm partitions a set in two subsets of equal-size and with minimal cut. A simple heuristic is applied to obtain all the necessary partitions, as Algorithm 3 shows. The heuristic initially applies the Kernighan-Lin algorithm to the graph G. It then recursively applies the Kernighan-Lin algorithm on the subsets, until all the $|T|$ threads have been partitioned in equally sized $|N|$ subsets, where $|T|$ and $|N|$ represent the number of threads in the application and the number of nodes in the machine respectively. This method does not guarantee that the sum of the weights of the edges crossing the $|N|$ components will be minimal. Fortunately the optimal solution is not needed: given that the similarity graph is already based on approximate measurements (due to sampling), a compromise is made by using this approximate method as a heuristic to what would be the optimal partitioning.

Algorithm 3: Recursive Kernighan-Lin

```
  Input: $G = (V, E)$, a similarity graph
1 if $|V| == |N|$ then return ;
2 $G_1, G_2 \leftarrow$ Kernighan-Lin(G)
3 Recursive-KL($G_1$)
4 Recursive-KL($G_2$)
```

After threads have been partitioned, PTB must decide in which node each partition will be allocated. The goal is to keep as many threads as possible with their current node affinity and thus avoid unnecessary migrations. To do that the greedy algorithm shown in Algorithm 4 is employed. This heuristic works by, for each unallocated thread cluster s, finding an available node n that already has as many threads of s as possible. It then migrates all threads of s that were not already in n to this new node and restart, with a new s, until all thread clusters have been migrated.

Algorithm 4: Thread cluster allocation

```
Input: $N, S$
$N' \leftarrow N$
forall $s \in S$ do
    max_counter $\leftarrow -1$
    max_node $\leftarrow$ undef
    forall $n \in N'$ do
        counter $\leftarrow 0$
        forall $t \in s$ do
            if CurrentNode($t$) $==$ $n$ then
                counter++
            end
        end
        if counter $>$ max_counter then
            max_counter $\leftarrow$ counter
            max_node $\leftarrow n$
        end
    end
    $N' \leftarrow N' \setminus \{n\}$
end
```

5.3 Page distribution

The third phase of PTB is page distribution. This is done immediately after threads have been migrated. A naive approach could migrate pages to the node that contains threads that access them the most. However, this is not optimal because it could considerably increase the congestion to node's memory controllers and NUMA interconnections, as detailed in Section 4.3. To avoid that, PTB uses a strategy that keeps track of the bandwidth usage at each node as pages are allocated. When the bandwidth usage reaches the maximum bandwidth allowed for the nodes, the remaining pages are interleaved across the other nodes to allow for an even congestion distribution. This approach is detailed below.

PTB's page distribution algorithm (Algorithm 5) works at the node level. Its first step is to estimate a node's bandwidth to each page (lines 7-10 and Algorithm 6). Then, it iterates over all pages, until either every page has been assigned to a node or until the bandwidth of all nodes has been "consumed" (line 13) – in which case interleaved

Algorithm 5: Page distribution algorithm

```
Input: $P, N, D_t$
Output: $D_p$
forall $n \in N$ do
    forall $n' \in N$ do
        $B_n[n'] \leftarrow$ MaxBandwidth$(n, n')$
    end
end
$D_p \leftarrow \emptyset$
forall $p \in P$ do
    forall $n \in N$ do
        $b_n[p] \leftarrow$ BandwidthFromNodeToPage$(n, p)$
    end
end
forall $p \in P$ do
    if $P == \emptyset$ or $\sum_n \sum_{n'} B_n[n'] == 0$ then
        Break;
    end
    if $\sum_{t \in T} A_t[p] \leq$ Min_Acc then
        skip page;
    end
    $n^* \leftarrow$ undef
    $\mathcal{B}_{n^*}[p] \leftarrow -\infty$
    forall $n \in N$ do
        $\mathcal{B}_n[p] \leftarrow$ NodeScore$(n, p)$
        if $\mathcal{B}_n[p] > \mathcal{B}_{n^*}[p]$ then
            $n^* \leftarrow n$
            $\mathcal{B}_{n^*}[p] \leftarrow \mathcal{B}_n[p]$
        end
    end
    $D_p \leftarrow D_p \cup \{(p, n^*)\}$
    forall $n \in N$ do
        $B_{n^*}[n] \leftarrow \max(B_{n^*}[n] - b_n[p], 0)$
    end
    $P \leftarrow P \setminus \{p\}$
end
forall $p \in P$ do
    $D_p \leftarrow D_p+$ interleaved distribution
end
```

distribution will take place (lines 34-36). To prevent pages with only a few accesses from being migrated, a page is skipped if it has less than `Min_Acc` accesses (total) from all threads (lines 16-18).

Each page can potentially be migrated to any node. A score (Algorithm 7) is computed for each node and the node with the highest score is chosen as the page's destination (lines 21-27). The algorithm then adds this `(page, destination_node)` pair to a set for later migration (line 28). The final step is to deduct the estimated used bandwidth for that page from all nodes (lines 29-31).

If all nodes' bandwidth is exhausted but there are still pages to be processed, it uses a simple interleave policy to minimize congestion (lines 34-36).

Algorithm 6 shows how bandwidth is calculated: node n's bandwidth to page p is estimated as the total number of accesses from threads in n to p, multiplied by the size of each access (the cache line size `ls`) and divided by the time spent in the last PTB cycle (τ).

Algorithm 6: BandwidthFromNodeToPage(n, p)

```
  Input: $p, n, \tau, A, D_t$
  Output: $b_n[p]$
1 $b_n[p] \leftarrow \sum_{(t,n)\in D_t} A_t[p]$
2 $b_n[p] \leftarrow b_n[p] \cdot ls/\tau$
3 return $b_n[p]$
```

A node is a good candidate to receive a page if it has enough spare bandwidth available to all nodes that try to access that page frequently. Function `NodeScore` (Algorithm 7) attempts to capture that. Consider deciding if n is a good placement for page p. If node n' does not access page p much (low $b_{n'}[p]$), then n' should not dictate whether n is a good candidate or not (it would be irrelevant for n'). If n does not have much bandwidth left to n' (low $B_n[n']$), then placing p in n would risk congesting n. Thus, this metric would favor configurations in which both $B_n[n']$ and $b_{n'}[p]$ are high, that is, node n has bandwidth to spare available to n' and n' accesses p frequently. Finally, page migration is an expensive operation, and thus a mechanism to minimize page bouncing between nodes is employed. To do that, the score $\mathcal{B}_n[p]$ is multiplied by a correctional factor $\mathtt{C}_2$ if n is the page's current node. This correction mechanism, just like the one used in the thread case, has the goal of overcoming small imbalances that would otherwise lead to a competition for that page, and thus bouncing.

Algorithm 7: NodeScore(n, p)

```
  Input: $n, p, b, B$
  Output: $\mathcal{B}_n[p]$
1 $\mathcal{B}_n[p] \leftarrow \sum_{n'\in N} B_{n'}[n] \cdot b_{n'}[p]$
2 if CurrentNode($p$) $==$ $n$ then
3     $\mathcal{B}_n[p] \leftarrow \mathtt{C}_2 \cdot \mathcal{B}_n[p]$
4 end
5 return $\mathcal{B}_n[p]$
```

Chapter 6

A Performance Study of PTB

This experimental evaluation of PTB was performed on the machine described in Chapter 4 and it uses benchmarks from the Parsec version 3.0 [7], NAS Parallel Benchmark (NPB) version 3.3.1 [6] and Metis [35] suites. The evaluation uses all programs, except for the following programs from the Parsec suite: `dedup`, which failed to compile; `vips`, which crashes under the default Linux scheduler; and `x264`, whose execution time is too short and no parameterization on the input size was possible. Some programs in the benchmark suites execute too fast even with the largest available dataset. Therefore, for this experimental evaluation, whenever possible execution parameters were changed to increase execution time. All programs and parameter/input sizes used are listed in Table 6.1. For each experimental data point in the study the program was run at least 10 times and always with 64 threads.

The execution of each benchmark is divided into an initialization phase, a Region Of Interest (ROI) and a termination phase. The ROI is the part of the benchmark that is timed and used to evaluate speedups. A program's initialization phase sometimes has a similar behavior to its ROI, but this part of the program is not timed in the evaluation. The issue is that if PTB were to be enabled during initialization, the program could enter the ROI with threads and pages already placed in optimal nodes, which would hide PTB's migration costs. Thus, to make the measurement fair, the programs are modified to communicate to PTB when they enter/exit the ROI. Benchmarks from the Parsec suite already specify the ROI explicitly. For all other benchmarks, the code was manually inspected to identify the timer's start/stop calls that define the ROI, after which a call to PTB's enable/disable function was inserted. There is a single continuous ROI in each benchmark.

For the rest of this chapter a list of questions is provided to guide the reader through the main results of the experiments and their corresponding analyses.

6.1 What is PTB overhead?

The overhead of PTB comes from three different sources: profiling overhead, calculation overhead and migration overhead. The profiling overhead is the cost of stopping the program to obtain a measurement and depends on the sampling rate. The calculation

Table 6.1: Programs and parameters/input size used

Program	Suite	Parameters/Input
blackscholes	Parsec	native, with 100M input
bodytrack	Parsec	native
canneal	Parsec	native
facesim	Parsec	native
ferret	Parsec	native
fluidanimate	Parsec	native
freqmine	Parsec	native
raytrace	Parsec	native, with `-frames 1000`
streamcluster	Parsec	native
swaptions	Parsec	native, with `-sm 10000000`
bt	NPB	C
cg	NPB	C
ep	NPB	C
ft	NPB	C
is	NPB	D
lu	NPB	C
mg	NPB	C
sp	NPB	C
ua	NPB	C
wrmem	Metis	`-q -s 1000`
kmeans	Metis	`30 256 50000000 80 -q`
pca	Metis	`-R 2048 -C 100 -q`
matrixmult	Metis	`-l 8192 -q`
wc	Metis	10^6 keys
wr	Metis	800 MB

overhead is mostly the cost of accessing the hash table and is dependent on the size of that table. The migration overhead is the cost of migrating pages and threads.

The profiling and calculation overheads are measured by running a program with the PTB module loaded, but with migrations disabled. Thread migration, on one hand, is relatively cheap, other than for potential cache effects. Page migration, on the other hand, can be expensive [33]. Measuring the migration overhead is tricky because it is hard to separate the overhead of doing the migration from the speedup that results from migrating. Since PTB uses standard system calls to perform page and thread migrations, this evaluation does not measure the migration overhead.

Different profiling rates and hash sizes were tested to measure the profiling and calculation overheads. The results of these experiments are illustrated in Figure 6.1 and in the heatmap of Figure 6.2 for the benchmark `mg`.

Figure 6.1 shows how much overhead can be accredited to the following: profiling, the thread phase and the page phase. The x-axis shows different `profiling rate-hash size` configurations. The *program* bar corresponds to the program running without PTB. Each of the three other bars represent the time increase due to adding the corresponding feature. As one can see, profiling (in purple) has a low overhead with different profiling

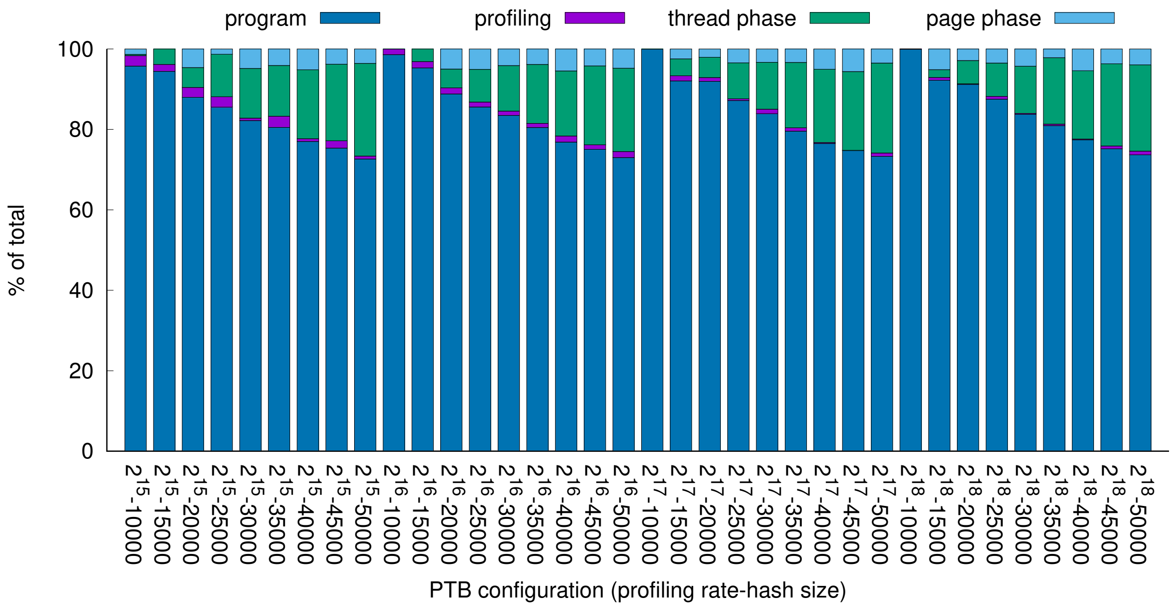


Figure 6.1: PTB overhead separated by phase (for benchmark `mg`)

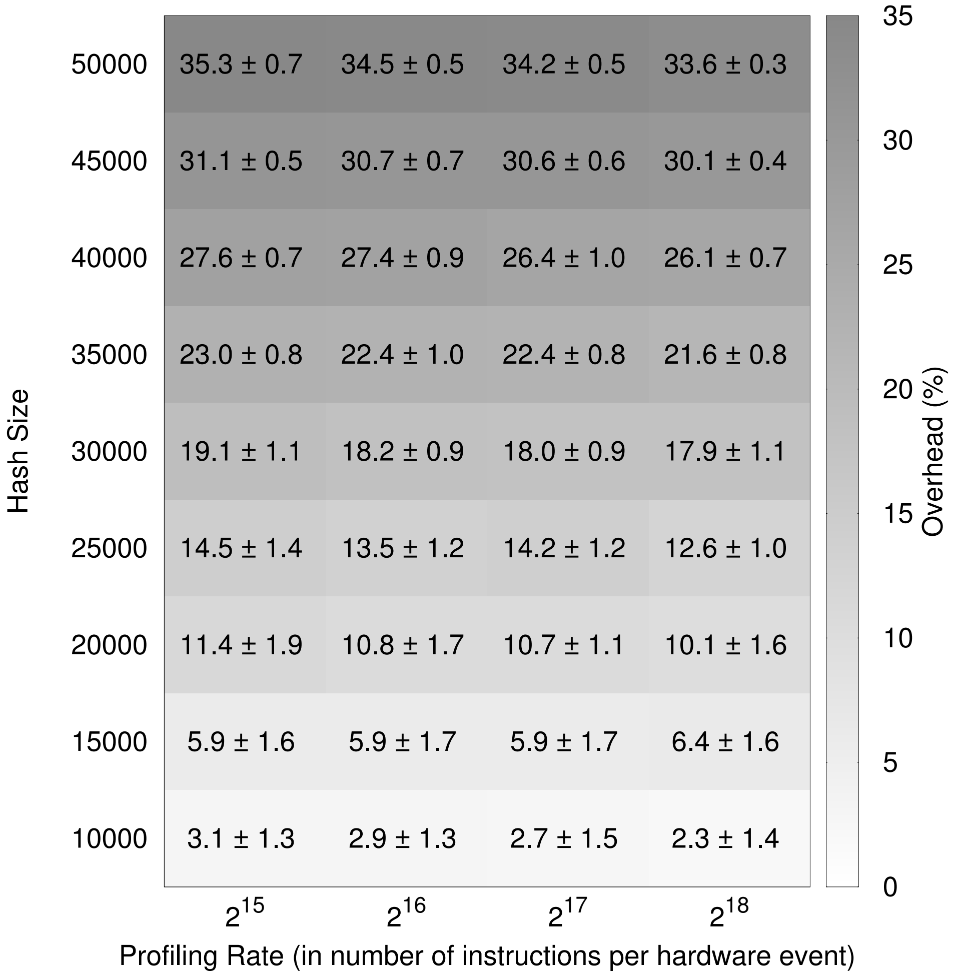


Figure 6.2: PTB overhead for `mg` (in % of running time)

rate and hash sizes. The page phase (in light blue) has a higher impact, but it is also constant with different configurations. The thread phase, on the other hand, increases considerably as the hash size increases. This can be easily explained by looking back at Equation 5.1: cosine similarity has complexity $O(|P|)$ and it has to be calculated for all pairs of threads (which means traversing the memory histograms multiple times).

Figure 6.2 shows the overhead in a different manner, as a heatmap. In this figure, both the number in each cell and its shade are the profiling and calculation overheads, given as a percentage of the program's running time under Linux's default scheduler. Darker backgrounds indicate higher overhead.

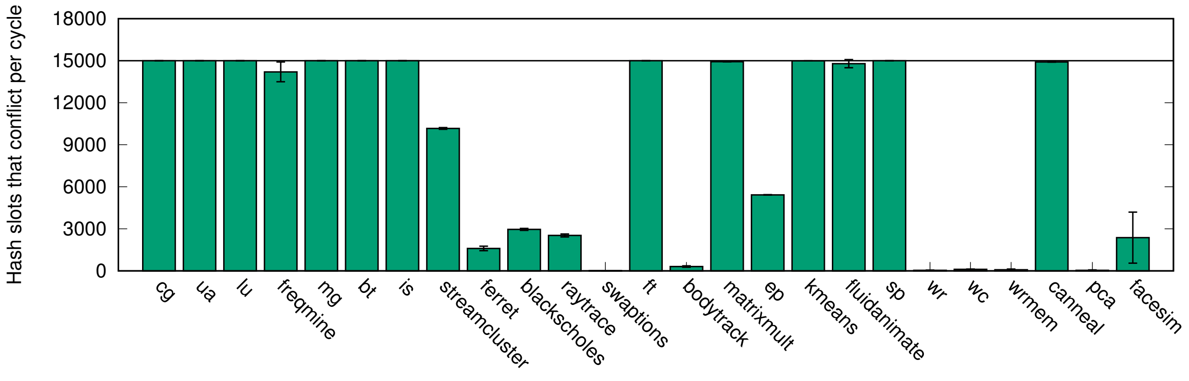
As expected, for a given hash size, the overhead decreases as the number of instructions between hardware events increases (profiling less often). However, the difference is numerically small and not statistically significant. Thus, the remainder of the experimental evaluation uses a profiling rate that samples every 2^{15} instructions.

On the other hand, the overhead increases significantly as the size of the hash table increases, thus indicating that the calculation overhead is dominant. The size of the hash table directly influences the accuracy of the estimations of bandwidth use in PTB — a larger hash is better. Hashes with sizes greater or equal to 20000 have over 10% of overhead, which is too large. A hash of size 15000 keeps the overhead at 6%, which is a good compromise between accuracy and speed. Thus, for the remainder of the experimental evaluation, this was the hash size chosen.

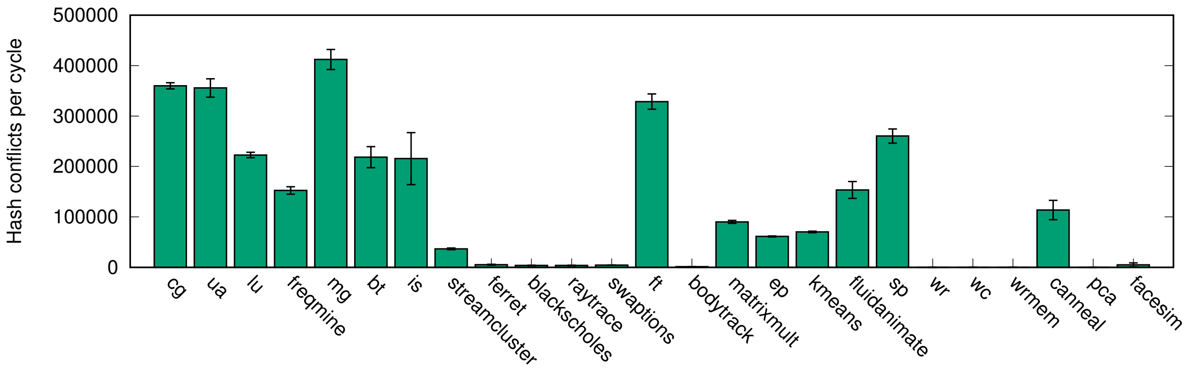
6.2 How hash conflicts impact accuracy and performance?

As stated in Section 5.1, the hash table used to store accesses trades accuracy for speed. To evaluate this tradeoff, the number of slots in which there was a conflict and the total number of conflicts, per cycle, were measured. Results are in Figure 6.3. The first graph shows the number of slots that had at least one conflict per cycle. The second graph (notice the logarithmic scale) shows the total number of conflicts per cycle. In all programs, all hash slots were occupied (recorded at least one access).

Correlating the data in Figure 6.3 with the speedups reported in Section 6.4 reveals that benchmarks with low number of conflicts usually have good speedups (`wr`, `wc`, `pca` and `facesim`). However, `sp`, `wrmem` and `canneal`, that also had good speedups with PTB, had a relatively high number of conflicts. The majority of programs with high number of conflicts (`cg`, `ua`, `lu`, `freqmine`, `mg`, `bt` and `is`) do not perform as well. Therefore a potential improvement for PTB would be to attempt different hashing strategies to reduce conflicts or to reevaluate the handling of conflicts in the hash data structure.



(a) Hash slots that conflict per cycle



(b) Hash conflicts per cycle

Figure 6.3: Conflicts in the hash table

6.3 Parameter tuning

Besides hash size and sampling frequency, PTB uses three other parameters, namely $\texttt{C}_1$, $\texttt{C}_2$ and `Min_Acc`. These parameters may need to be tuned if PTB is to be used in different machines. This section discusses how the values were chosen in the experiments and what should be kept in mind when adjusting them.

$\texttt{C}_1$ and $\texttt{C}_2$ are the multiplicative constants used in the similarity-index calculation and `NodeScore` function, respectively. The higher their values, the higher is the incentive given to keep threads/pages on their current nodes. Thus, in machines with high migration overheads, to minimize migrations and to only do so when there is a notorious improper placement larger values of $\texttt{C}_1$, $\texttt{C}_2$ should be used.

`Min_Acc` is the minimum total number of accesses that a page must receive to be considered for migration. In machines in which page migration overhead is higher and/or remote latency is not much higher than local latency, using higher values of `Min_Acc` is advisable.

To select appropriate values for these parameters, several numbers in a wide range were tested, for a select group of benchmarks. Parameters were tested individually, *i.e.* two parameters were fixed while the third was varied. $\texttt{C}_1$ and $\texttt{C}_2$ were tested in the range $[0.1, 32]$ and `Min_Acc` was tested in the range $[1, 32]$. Results are shown in Figure 6.4 (note that, for each benchmark, a different scale is used).

Benchmark `raytrace` (Figure 6.4a) shows no sensibility to any of the parameters and its speedup remains very close to 1 in all parameter configurations.

In benchmark `canneal` (Figure 6.4b) despite the numerical difference in speedup in the graphs for $\texttt{C}_1$ (6.4b.i) and `Min_Acc` (6.4b.iii), no trend can be observed and the error bars make all results statistically equivalent – except for value 32 of `Min_Acc`, which shows performance loss. As for $\texttt{C}_2$, values closer to 1 perform better.

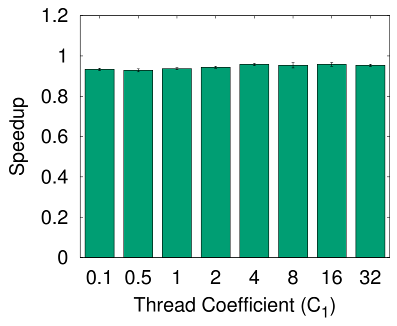
`cg` (Figure 6.4c) is $\texttt{C}_1$-insensitive, but it shows opposite behavior than `canneal` regarding $\texttt{C}_2$ and `Min_Acc`: values equal or higher than 2 for $\texttt{C}_2$ and $\texttt{Min\_Acc} = 32$ perform better (note, however, that for all configurations, `cg` loses performance).

Finally, `mg` (Figure 6.4d) shows similar behavior to `cg`: Values of 2 or higher for $\texttt{C}_2$ perform better, and equivalent performance regardless of the values used for $\texttt{C}_1$ and `Min_Acc`.

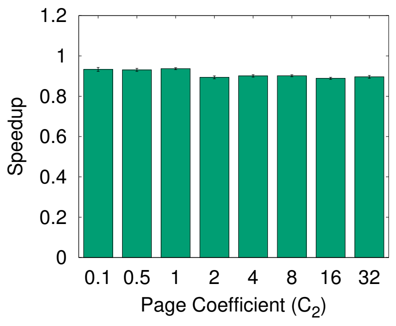
This analysis indicates that PTB is insensitive to the thread coefficient $\texttt{C}_1$. Hence, the value 1 is used in the experimental evaluation.

$\texttt{C}_2$ shows conflicting behavior, as benchmarks have opposing trends. A value between 1 and 2 seems to be the turning point between the tested benchmarks – thus, a value of 1.5 is used.

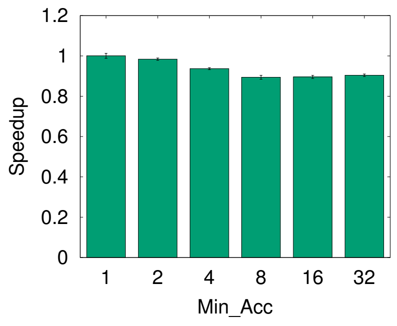
`Min_Acc` also shows opposing results between benchmarks, although not as prominent as in $\texttt{C}_2$. A midway compromise is made by using 16.



(a.i) `C`$_1$

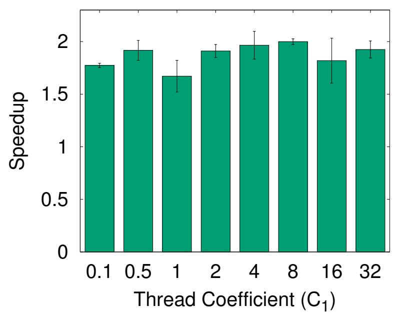


(a.ii) `C`$_2$

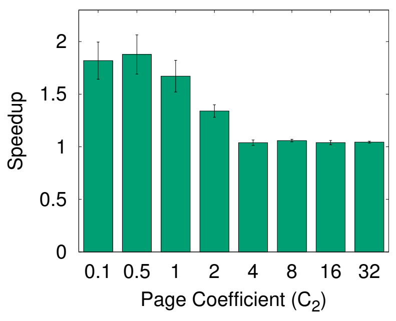


(a.iii) `Min_Acc`

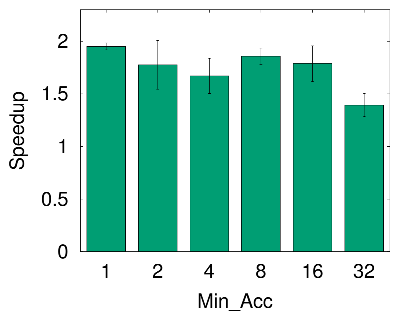
(a) `raytrace`



(b.i) `C`$_1$

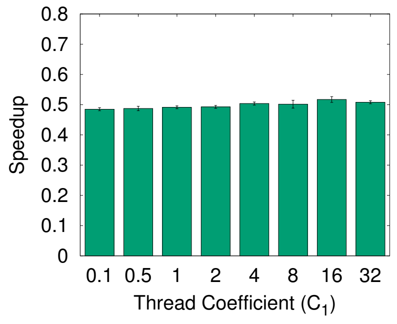


(b.ii) `C`$_2$

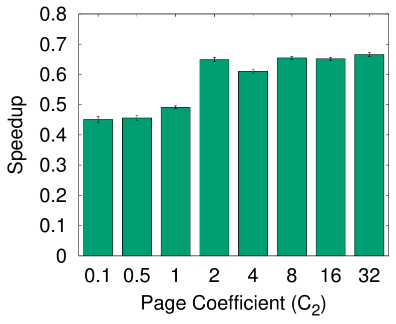


(b.iii) `Min_Acc`

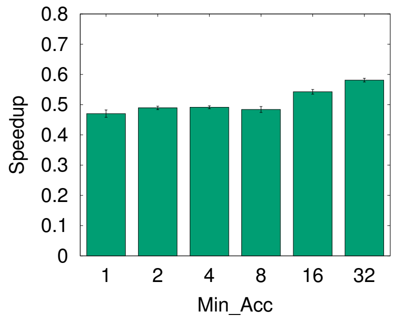
(b) `canneal`



(c.i) `C`$_1$

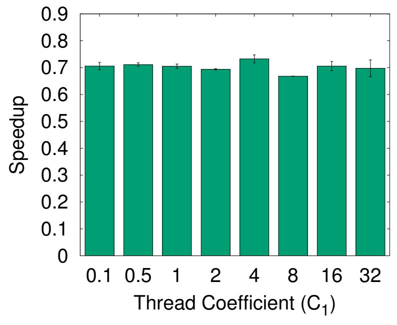


(c.ii) `C`$_2$

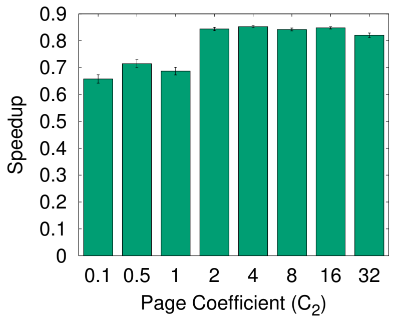


(c.iii) `Min_Acc`

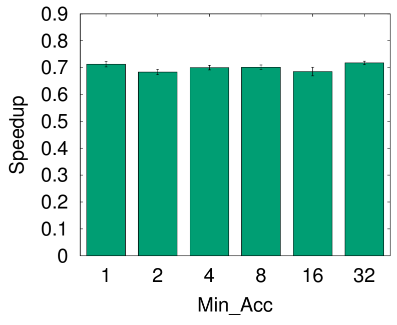
(c) `cg`



(d.i) `C`$_1$



(d.ii) `C`$_2$



(d.iii) `Min_Acc`

(d) `mg`

Figure 6.4: Parameter tuning

6.4 How does PTB compare to alternative solutions?

This evaluation compares PTB with three other techniques. The baseline for comparison is the default Linux 3.9 scheduler. The second solution is a pinned-thread version of the 3.9 Linux kernel (*Fixed Affinity*). Pinning thread by affinity can increase cache-sharing and program performance [36]. For OpenMP-based benchmarks, pinning is done by setting the `GOMP_CPU_AFFINITY` environment variable to `0-63`: thread 0 is pinned to core 0, thread 1 to core 1, *etc.* For pthreads-based benchmarks, the applications were modified to manually set thread affinities. The third source for comparison is Linux's automatic NUMA balancing in kernel version 3.11 [16] (*NUMA Balancing*). Unfortunately, it was not possible to test alternative solutions presented in the literature, as there was no access to the required source code. In some cases, only part of the required code was available. In others, paper authors were contacted but they replied that the code was not available. Since previous work was evaluated on different experimental platforms, reported results can not be directly compared.

While PTB is only enabled when the application enters the ROI, all alternative techniques are enabled since program startup. Thus, in their cases, migrations may take place during program initialization and these migrations costs are hidden from the measured speedup.

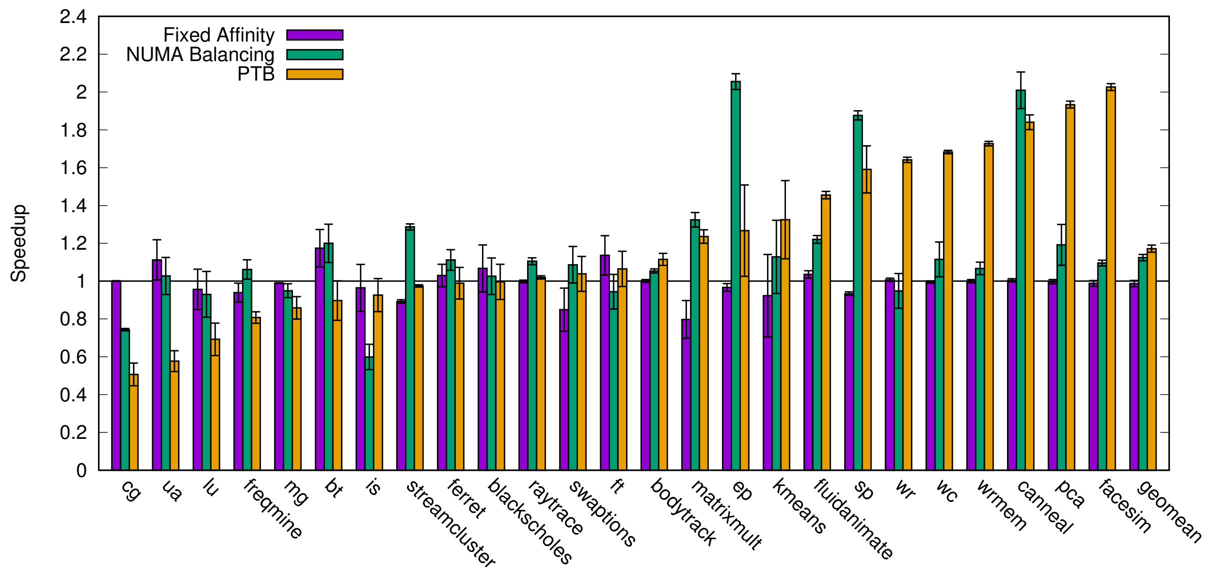


Figure 6.5: Speedup and comparison to other techniques

Figure 6.5 shows the speedups for all benchmarks, as well as the geometric mean. PTB produces (statistically significant) speedups in 10 applications, while slowing down 6 – the remaining 10 benchmark speedups are statistically equivalent to 1. For the 25 benchmarks, Fixed Affinity was better than the other two competing solutions (NUMA Balancing and PTB) only in one of them (`cg`) – still, with a speedup of 1 (the other solutions slowed this application down). NUMA Balancing "won" on 6 benchmarks and PTB was the best choice in 7 others. The remaining 11 benchmarks had two or more solutions with statistically equivalent speedups. On 7 applications, PTB produces speedups ranging

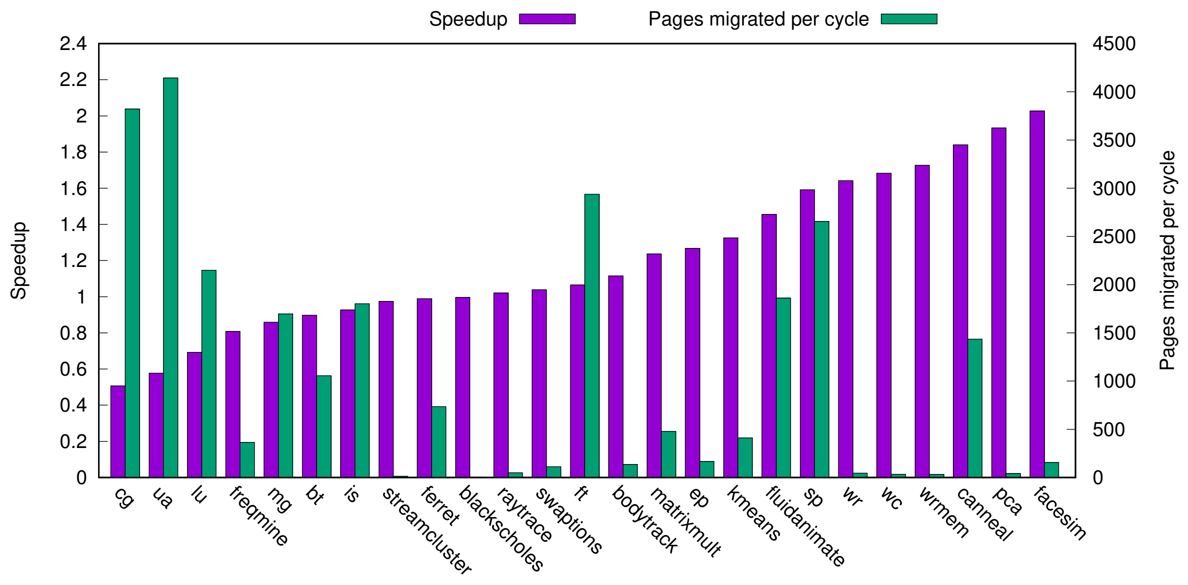


Figure 6.6: Speedup and number of pages migrated per cycle

from 1.6x to 2x when NUMA Balancing stayed below 1.2x. or resulted in slowdowns.

The slowdowns produced by PTB can be partially explained by Figure 6.6, which shows the average number of pages migrated per PTB cycle. For most benchmarks that have a slowdown with PTB, the number of pages migrated is high, whereas applications with good speedups usually had a small number of page migrations. Because page migration is an expensive task, too much time is being spent on this task, thus slowing the application down.

Another explanation for the slowdowns can be made by correlating the speedups with the graphs shown in Section 6.2. In all benchmarks in which PTB had a slowdown, all hash slots conflicted in every PTB cycle. These conflicts reduce accuracy and could be causing wrong migrations.

However, it is important to note that a high number of conflicts not necessarily means that there will be performance loss – in some cases, it is quite the opposite: benchmarks `canneal`, `sp`, `fluidanimate`, `kmeans` and `matrixmult` had significant speedups even with 100% of hash conflicts. Some possible explanations for this phenomenon are:

(a) the number of hits in each slot was not high enough to pass the `Min_Acc` threshold for all hash slots. Notice, on Figure 6.3b, that the total number of conflicts for some of these benchmarks is smaller.

(b) Pages that conflict are accessed by the same (few) threads. Hence, even though the count on each page is not correct, the set of threads that access them is. Thus, there would be no migrations done to the wrong node.

(c) Pages are shared by all threads – thus, interleaved allocation is indicated and conflicts would lead to interleaving.

The execution time of some benchmarks (`blackscholes`, `swaptions`, `kmeans`, `bt`, `ft`, `is`, `lu` and `ua`) had significant variability (greater than 10%) even when running with Linux's default scheduler, hence their larger standard deviation on Figure 6.5. This variability may be due to the use of random methods (such as in `swaptions`) and/or because of different initial allocations: remember that Linux, by default, uses first touch allocation, so if threads were created in different nodes, the memory layout will also be different.

Chapter 7

Conclusion

This work introduces PTB, a page, thread and bandwidth allocation approach that seeks to balance bandwidth utilization by dynamically migrating threads and memory pages between processors. PTB combines an automatic (offline) profile machine characterization procedure with application online profiling. The experimental evaluation has confirmed intuitions that motivated the design of PTB: matching the traffic generated by remote memory page accesses with the available bandwidth in the machine is important, and so is alleviating congestion in memory controllers. A surprising insight of this work is that creating a memory page and thread distribution that favors local access actually comes third in the list of priority for such algorithms. For a number of benchmarks PTB produces speedups ranging from 1.6x to 2x that, when compared to Linux NUMA Balancing (limited to speedups of 1.2x), demonstrate the effectiveness of this approach.

Future research for PTB include:

- **The tradeoff between clustering threads based only in their orientations vs. based on the amount of shared accesses**
  Using cosine similarity clusters threads based only on the orientation of their memory accesses (in the vector representation). This could induce threads with a small amount of memory accesses, but with similar distributions, to be clustered together, instead of threads that share a high number of accesses to a few pages, but not to all (and thus have very different orientations). The question thus, is: what is more important, similar memory access patterns or the amount of shared accesses?

- **Dynamically turning PTB on/off**
  If the program has a regular behavior and if pages and threads have already been migrated and are in their optimal placements, PTB could be dynamically turned off or adjusted to incur less overhead.

- **Dynamically adjust PTB's parameters**
  As shown in Section 6.3, there is no universally optimal value for parameters $\texttt{C}_2$ and `Min_Acc`. Thus, an improvement on PTB would be a metric that automatically and dynamically sets the appropriate value for these parameters.

- **Relationship between page migrations and slowdown**
  As shown in Section 6.4, benchmarks that lost performance with PTB had a high

count of pages migrated, an expensive operation. Thus, a technique that decrease these migrations should yield a positive impact on such benchmarks;

- **Offline distribution**
  For the set of applications that have a regular access pattern and whose memory layout is the same regardless of input, an offline distribution algorithm would be a good improvement. Using PTB's thread clustering and page distribution heuristics, it would be possible to determine an optimal initial allocation, so that there would be no migration overhead – in fact, PTB could actually be disabled after the initial allocation.

- **Different hashing strategies**
  As shown in Section 6.2, there is a correlation between a program losing performance and a high number of conflicts in the hash table. Hence, strategies that reduce conflicts or reevaluating the handling of conflicts in the hash data structure could lead to improved results.

Bibliography

[1] Documentation for /proc/sys/kernel/*. `https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/sysctl/kernel.txt?id=refs/tags/v3.17-rc7#n428`.

[2] Linux Programmer's Manual: numa - NUMA policy library. `http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm`.

[3] Advanced Micro Devices. *Bios and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors*, January 2013.

[4] Jennifer Anderson and Monica Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Special Interest Group on Programming Languages (SIGPLAN)*, pages 112–125, Albuquerque, New Mexico, USA, 1993.

[5] Joseph Antony, Pete Janes, and Alistair Rendell. Exploring thread and memory placement on NUMA architectures: Solaris and linux, ultrasparc/fireplane and opteron/hypertransport. In *High Performance Computing (HiPC)*, pages 338–352, Bangalore, India, 2006.

[6] David Bailey, Eric Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical report, 1994.

[7] Christian Bienia. *Benchmarking Modern Multiprocessors.* PhD thesis, Princeton University, January 2011.

[8] Sergey Blagodurov, Sergey Zhuravlev, Alexandra Fedorova, and Ali Kamali. A case for NUMA-aware contention management on multicore systems. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 557–558, Vienna, Austria, September 2010.

[9] Robert Blumofe, Christopher Joerg, Bradley Kuszmaul, Charles Leiserson, Keith Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55 – 69, 1996.

[10] François Broquedis, Nathalie Furmento, Brice Goglin, Pierre-André Wacrenier, and Raymond Namyst. ForestGOMP: An efficient OpenMP environment for NUMA architectures. *International Journal of Parallel Programming (IJPP)*, 38(5):418–439, 2010.

[11] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. *Special Interest Group on Programming Languages (SIGPLAN)*, 29(11):12–24, November 1994.

[12] Pat Conway, Nathan Kalyanasundharam, Gregg Donley, Kevin Lepak, and Bill Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *IEEE Micro*, 30(2):16–29, March 2010.

[13] Julita Corbalan, Xavier Martorell, and Jesus Labarta. Evaluation of the memory page migration influence in the system performance: The case of the sgi o2000. In *International Conference on Supercomputing (ICS)*, pages 121–129, San Francisco, CA, USA, 2003.

[14] Jonathan Corbet. A potential NUMA scheduling solution. `https://lwn.net/Articles/522093/`, October 2012.

[15] Jonathan Corbet. NUMA in a hurry. `https://lwn.net/Articles/524977/`, November 2012.

[16] Jonathan Corbet. NUMA scheduling progress. `https://lwn.net/Articles/568870/`, October 2013.

[17] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quéma, and Mark Roth. Traffic management: a holistic approach to memory placement on numa systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 381–394, Houston, Texas, USA, March 2013. ACM.

[18] Mathias Diener, Eduardo Cruz, and Philippe Navaux. Communication-based mapping using shared pages. In *International Parallel & Distributed Processing Symposium (IPDPS)*, pages 700–711, Boston, Massachusetts, USA, May 2013.

[19] Paul Drongowski. Instruction-based sampling: A new performance analysis technique for AMD family 10h processors. Technical report, Advanced Micro Devices, Inc., 2007.

[20] Olivier Goldschmidt and Dorit Hochbaum. Polynomial algorithm for the k-cut problem. In *Foundations of Computer Science*, pages 444–451, October 1988.

[21] Mel Gorman. Basic scheduler support for automatic NUMA balancing v8. `http://thread.gmane.org/gmane.linux.kernel/1568976`, September 2013.

[22] Lei Huang, Haoqiang Jin, Liqi Yi, and Barbara Chapman. Enabling locality-aware computations in openmp. *Scientific Programming*, 18(3-4):169–181, 2010.

[23] Intel Corporation. *Intel 64 and IA-32 Architecture Software Developer's Manual, Volume 3B*, April 2016.

[24] Dongming Jiang and Jaswinder Singh. Scaling application performance on a cache-coherent multiprocessors. In *International Symposium on Computer Architecture (ISCA)*, pages 305–316, May 1999.

[25] Laxmikant Kale and Sanjeev Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Conference on Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 91–108, Washington, D.C., USA, 1993.

[26] Ali Kamali. Sharing aware scheduling on multicore systems. Master's thesis, Simon Fraser University, 2010.

[27] Brian Kernighan and Shen Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, February 1970.

[28] Andi Kleen. A NUMA API for LINUX. Technical report, April 2005.

[29] Baptiste Lepers, Vivien Quema, and Alexandra Fedorova. Thread and memory placement on NUMA systems: Asymmetry matters. In *USENIX Annual Technical Conference*, pages 277–289, Santa Clara, California, USA, July 2015. USENIX Association.

[30] Hui Li, Sudarsan Tandri, Michael Stumm, and Kenneth Sevcik. Locality and loop scheduling on NUMA multiprocessors. *International Conference on Parallel Processing (ICPP)*, 2:140–147, 1993.

[31] Henrik Löf and Sverker Holmgren. Affinity-on-next-touch: Increasing the performance of an industrial pde solver on a CC-NUMA system. In *International Conference on Supercomputing (ICS)*, pages 387–392, Cambridge, Massachusetts, 2005.

[32] Zoltan Majo and Thomas Gross. Memory system performance in a NUMA multicore multiprocessor. In *International Systems and Storage Conference (Systor)*, pages 12:1–12:10, Haifa, Israel, 2011.

[33] Zoltan Majo and Thomas Gross. Matching memory access patterns and data placement for NUMA systems. In *International Symposium on Code Generation and Optimization (CGO)*, pages 230–241, San Jose, California, 2012.

[34] Zoltan Majo and Thomas Gross. (Mis)Understanding the NUMA Memory System Performance of Multithreaded Workloads. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 11–22, 2013.

[35] Yandong Mao, Robert Morris, and Frans Kaashoek. Optimizing mapreduce for multicore architectures. Technical report, MIT, 2010.

[36] Abdelhafid Mazouz, Sid-Ahmed-Ali Touati, and Denis Barthou. Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures. In *International Conference on High Performance Computing & Simulation (HPCS)*, pages 273–279, July 2011.

[37] John McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[38] Rada Mihalcea, Courtney Corley, and Carlo Strapparava. Corpus-based and knowledge-based measures of text semantic similarity. In *National Conference on Artififical Intelligence (AAAI)*, volume 6, pages 775–780, Boston, Massachusetts, USA, 2006.

[39] Ozcan Ozturk. Data locality and parallelism optimization using a constraint-based approach. *Journal of Parallel and Distributed Computing*, 71(2):280–287, February 2011.

[40] Stephen Park and Keith Miller. Random number generators: Good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, October 1988.

[41] David Patterson and John Hennessy. *Computer Organization and Design: The Hardware/Software Interface.* Morgan Kaufmann, San Francisco, CA, USA, 5th edition, 2013.

[42] Guilherme Piccoli, Henrique Santos, Raphael Rodrigues, Christiane Pousa, Edson Borin, and Fernando Quintão Pereira. Compiler support for selective page migration in NUMA architectures. In *Parallel Architectures and Compilation Techniques (PACT)*, pages 369–380, Edmonton, AB, Canada, 2014.

[43] Laércio Pilla, Christiane Pousa Ribeiro, Daniel Cordeiro, and Jean-François Mehaut. Charm++ on NUMA Platforms: the impact of SMP Optimizations and a NUMA-aware Load Balancing. In *Workshop of the INRIA-Illinois Joint Laboratory on Petascale Computing*, Urbana, United States, 2010.

[44] Christiane Ribeiro, Márcio Castro, Jean-François Méhaut, and Alexandre Carissimi. Improving memory affinity of geophysics applications on NUMA platforms using minas. In *High Performance Computing for Computational Science (VECPAR)*, pages 279–292, Berkeley, CA, 2011. Springer-Verlag.

[45] Kenneth Sevcik and Songnian Zhou. Performance benefits and limitations of large NUMA multiprocessors. *Performance Evaluation*, 20(1–3):185 – 205, May 1994.

[46] Per Stenström, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherent NUMA and COMA architectures. In *International Symposium on Computer Architecture (ISCA)*, pages 80–91, Queensland, Australia, 1992.

[47] Jorge Stolfi. Coord system ca 0.svg. `https://commons.wikimedia.org/wiki/File:Coord_system_CA_0.svg`, May 2009.

[48] Alexander Strehl, Er Strehl, Joydeep Ghosh, and Raymond Mooney. Impact of similarity measures on web-page clustering. In *National Conference on Artififical Intelligence (AAAI)*, pages 58–64, Austin, Texas, USA, 2000.

[49] David Tam, Reza Azimi, and Michael Stumm. Thread clustering: Sharing-aware scheduling on SMP-CMP-SMT multiprocessors. In *EuroSys*, pages 47–58, Lisbon, Portugal, 2007.

[50] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining.* Pearson, Boston, MA, USA, 1st edition, 2005.

[51] Christian Terboven, Dieter an Mey, Dirk Schmidl, Henry Jin, and Thomas Reichstein. Data and thread affinity in openmp programs. In *International Conference on Computing Frontiers*, pages 377–384, Ischia, Italy, May 2008.

[52] Rik van Riel. Automatic NUMA Balancing. `http://events.linuxfoundation.org/sites/events/files/slides/summit2014_riel_chegu_w_0340_automatic_numa_balancing_0.pdf`, April 2014.

[53] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. *Special Interest Group - Operating Systems Review (SIGOPS)*, 25(5):26–40, September 1991.

[54] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 279–289, Cambridge, Massachusetts, USA, 1996.

[55] Kenneth Wilson and Bob Aglietti. Dynamic page placement to improve locality in CC-NUMA multiprocessors for TPC-C. In *ACM/IEEE SC Conference*, pages 33–33, Denver, Colorado, November 2001.