INTERNATIONAL DOCTORAL
SCHOOL OF THE USC

Ruben
Laso Rodríguez

PhD Thesis

Dynamic workload optimisation on NUMA and heterogeneous architectures

Santiago de Compostela, 2023

Doctoral Programme in Information Technology Research

Ph.D. Thesis

DYNAMIC WORKLOAD OPTIMISATION ON NUMA AND HETEROGENEOUS ARCHITECTURES

Ruben Laso Rodríguez

ESCOLA DE DOUTORAMENTO INTERNACIONAL DA UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

PROGRAMA DE DOUTORAMENTO EN INVESTIGACIÓN EN TECNOLOXÍAS DA INFORMACIÓN

SANTIAGO DE COMPOSTELA
2023

Declaración do autor da tese

Don Ruben Laso Rodríguez
Título da tese: Dynamic workload optimisation on NUMA and heterogeneous architectures

Presento a miña tese, seguindo o procedemento adecuado ao Regulamento, e declaro que:

1. *A tese abarca os resultados da elaboración do meu traballo.*
2. *De ser o caso, na tese faise referencia ás colaboracións que tivo este traballo.*
3. *Confirmo que a tese non incorre en ningún tipo de plaxio doutros autores nin de traballos presentados por min para a obtención doutros títulos.*
4. *A tese é a versión definitiva presentada para a súa defensa e coincide a versión impresa coa presentada en formato electrónico.*

E comprométome a presentar o Compromiso Documental de Supervisión no caso de que o orixinal non estea na Escola.

En Santiago de Compostela, 4 de mayo de 2023

Asdo. Ruben Laso Rodríguez

Autorización do director/titor da tese

Dynamic workload optimisation on NUMA and heterogeneous architectures

Don José Carlos Cabaleiro Domínguez, Catedrático de Universidade da Área de Arquitectura e Tecnoloxía de Computadores da Universidade de Santiago de Compostela.

Don Tomás Fernández Pena, Catedrático de Universidade da Área de Arquitectura e Tecnoloxía de Computadores da Universidade de Santiago de Compostela.

INFORMAN:

Que a presente tese correspóndese co traballo realizado por ***Don Ruben Laso Rodríguez****, baixo a nosa dirección/titorización, e autorizamos a súa presentación, considerando que reúne os requisitos esixidos no Regulamento de Estudos de Doutoramento da USC, e que como directores/titores desta non incorre nas causas de abstención establecidas na Lei 40/2015.*
De acordo co indicado no Regulamento de Estudos de Doutoramento, declara tamén que a presente tese de doutoramento é idónea para ser defendida en base á modalidade de Monográfica con reproducción de publicaciones, nos que a participación do/a doutorando/a foi decisiva para a súa elaboración e as publicacións se axustan ao Plan de Investigación.

En Santiago de Compostela, 4 de mayo de 2023

Asdo. José Carlos Cabaleiro Domínguez
Director/a tese

Asdo. Tomás Fernández Pena
Director/a tese

No saber exactamente qué es 'El Plan' es parte de 'El Plan'.
— Fernando Alonso, *Formula 1.*

Do. Or do not. There is no try.
— Yoda, *Star Wars Episode V: The Empire Strikes Back.*

Dedications

To my advisors, Caba, Tomás and Fran, for their guidance and advice.

Acknowledgements

Thanks to my family for their continuous support and encouragement across all the stages of my education, career and life.

Thanks to Isabel, for her patience and understanding during the long hours I have spent working on this thesis.

Thanks to my colleagues at CiTIUS for their friendship, support and the long coffee breaks we have shared.

May 4, 2023

Additional acknowledgements

Thanks to Rafael Asenjo and the Department of Computer Architecture of Universidad de Málaga for sharing with us the source code of LogFit and for their valuable help.

Thanks to the people working at CESGA (https://www.cesga.es) for sharing with us anonymous data on the use of their systems, which allowed us to design some of the experiments included in this thesis.

This work has received financial support from the *Consellería de Cultura, Educación e Universidade* of *Xunta de Galicia* (accreditations 2016-2019, ED431G/08 and GRC-008, and 2019-2022 ED431G-2019/04, reference competitive group 2019-2021, ED431C 2018/19, and Grant Number GRC R2014/008) and the *Ministerio de Ciencia e Innovación* within the projects PID2019-104834GB-I00 and TIN2016-76373-P (AEI/FEDER, EU). It has been funded as well by networks R2016/045, R2016/037 and CAPAP-H and the European Regional Development Fund (ERDF), which acknowledges the CiTIUS—Centro Singular de Investigación en Tecnoloxías Intelixentes da Universidade de Santiago de Compostela—as a Research Center of the Galician University System.

Additionally, part of this work has been performed under Project HPC-EUROPA3 (INFRAIA-2016-1-730897), with the support of the EC Research Innovation Action under the H2020 Programme; in particular, the author gratefully acknowledges the support of Adrian Jackson (EPCC) and the computer resources and technical support provided by EPCC. Also, thanks to Catherine Inglis (EPCC) for her help with the logistics of my research stay at EPCC.

Contents

Resumo

Nesta tese afrontáronse desafíos na optimización dinámica da distribución e reparto de carga computacional en dous tipos diferentes de sistemas, particularmente en sistemas convencionais utilizando paralelismo heteroxéneo de CPU e GPU e en servidores *Non-Uniform Memory Access (NUMA)*. Cada tipo de sistema conta coas súas propias características e singularidades, de forma que requiren diferentes aproximacións e distintas solucións para os problemas que se abarcaron.

En primeiro lugar, abordouse o problema da asignación dinámica de carga de traballo en paralelismo heteroxéneo, combinando o poder de computación de CPUs e GPUs. Co auxe da computación de propósito xeral en GPUs, *general-purpose computing on Graphics Processing Units (GPGPU)*, novas oportunidades xurdiron no eido da computación paralela. Dende principios do século XXI, apareceron varias tecnoloxías para facilitar a programación neste tipo de procesadores, establecéndose *Compute Unified Device Architecture (CUDA)* como a mais popular. A principal vantaxe das GPUs reside na grande cantidade de núcleos de computación (na orde de miles no momento de escribir esta tese) e a estreita relación entre os devanditos núcleos e a súa memoria. Estas características fan que as GPUs sexan particularmente adecuadas nos problemas coñecidos como *embarrasingly parallel* e principalmente adicados ao cómputo de tipo aritmético, como se recolle no Capítulo 1. Inicialmente, foron códigos de simulación física e procesamento de imaxes os que máis se beneficiaron da GPGPU. Na actualidade, é no campo do *machine learning* onde se fai un maior uso destas arquitecturas que xa incorporan procesadores específicos (como os *tensor cores*) para este propósito. Sen embargo, utilizar soamente a GPU para a computación pode deixar a CPU ociosa, o que é subóptimo en termos de xestión dos recursos. Polo tanto, facer uso do paralelismo heteroxéneo é un camiño que merece ser explorado na busca dun mellor rendemento e a redución dos tempos de execución.

Co obxectivo de maximizar o uso da CPU e da GPU para acelerar a computación, particularmente en métodos iterativos ou con pasos de tempo, nesta tese proponse a librería *Iterative Heterogeneous Parallelism (IHP)*. Tal e como se explica no Capítulo 2, IHP divide o dominio global en dous subdominios que son asignados aos dous tipos de procesadores para acadar o seu obxectivo. O tamaño destes subdominios é recalculado periodicamente segundo o rendemento da CPU e da GPU, tal que, idealmente ambos

tipos de procesadores consumen o mesmo tempo na execución dos seus respectivos subdominios. Nesta tese desenvolvéronse dúas versións de IHP. A versión inicial de IHP, *IHP version 1 (IHPv1)*, asume un modelo de carga de traballo linear, onde o tempo de execución crece linearmente coa cantidade de traballo e o reparto do mesmo recalcúlase en función do tempo invertido pola CPU e pola GPU, respectivamente, para facer os cálculos que lles foron asignados na iteración anterior. A segunda versión, *IHP version 2 (IHPv2)*, engade varios modelos de rendemento, que son clasificados como linear, logarítmico e exponencial. Estes modelos son axustados coa serie histórica de datos de rendemento obtidos, de forma que o modelo que mellor axuste os tempos de execución previos utilízase para calcular o reparto de traballo a utilizar na seguinte iteración. Adicionalmente, IHPv2 inclúe mecanismos que utilizan os devanditos datos históricos de rendemento co fin de reducir as custosas transferencias de datos entre CPU e GPU. Desta forma os cambios súbitos no reparto de traballo entre ámbolos dous tipos de procesadores poden mitigarse, contando o algoritmo cunha maior estabilidade.

Para a avaliación destas propostas, fixéronse varias probas a partires de códigos de diferencias finitas utilizados no eido do procesamento de imaxes para reducir o ruído das mesmas. Estes códigos adáptanse á perfección ás características das GPUs dada a súa natureza masivamente paralela. Ademais, a súa sinxela estrutura permite inserir modificacións no código para emular distintos tipos de cargas computacionais e así avaliar o comportamento de IHPv1 e IHPv2 en diferentes situacións.

Os resultados recollidos no Capítulo 2 mostran que IHPv1 reduce os tempos de execución en cargas de traballo de tipo linear entre un 3.20 % e un 55 % en comparación ás implementacións que só utilizan a GPU. A porcentaxe de mellora varía dependendo en función tanto das características do código paralelo coma do tipo de rexistros que se utilicen para o cómputo, sendo os códigos máis intensivos onde as implementacións heteroxéneas teñen máis marxe de mellora. Noutros casos, o tempo de execución pode ser dominado polas transferencias entre CPU e GPU. Nótese que, inclusive nas peores situacións, onde a carga computacional é baixa, IHP consegue compensar o *overhead* e manter os tempos de execución por debaixo daqueles onde só se utiliza a GPU. Por outra banda, é coñecido que o rendemento das GPUs escala peor que o das CPUs ao operar con rexistros de punto flotante de dobre precisión. En consecuencia, as CPUs poden adquirir máis carga de traballo e facer unha maior contribución á mellora dos tempos de execución. En comparación con outras librerías, IHPv1 consegue obter un mellor rendemento grazas a un cálculo do reparto da carga de computación máis preto ao óptimo e á redución da cantidade de transferencias realizadas entre CPU e GPU.

Comparando as diferentes versións de IHP, IHPv2 consegue mellorar os resultados de IHPv1 ante diferentes cargas de traballo, tanto lineares como non lineares. Esta mellora é particularmente importante en cargas de traballo de tipo exponencial, dado que unha estratexia de "divide e vencerás" xa permite reducir a cantidade total de traballo realizado. É interesante comentar que IHPv1 é quen de acadar un bo resultado independentemente do tipo de *workload*. Isto débese a que a rexión na que se localiza o punto óptimo de reparto da carga de traballo pódese entender como localmente

linear. Polo tanto, unha vez que IHPv1 atopa valores próximos ao óptimo, o seu rendemento é comparable ao de IHPv2, que contempla diferentes tipos de carga. IHPv2 tamén consegue mellorar a IHPv1 lixeiramente nas transferencias de datos grazas aos mecanismos incorporados que recollen un histórico de rendemento e permiten reducir o tempo invertido nestas custosas comunicacións.

En canto ao consumo enerxético, a pesares das significativas reducións nos tempos de execución, as solucións que utilizan computación heteroxénea non conseguen mellorar as cifras das implementacións que soamente utilizan GPUs. Isto é debido á diferencia na eficiencia enerxética das CPUs en comparación ás GPUs en traballos masivamente paralelos, onde a relación entre rendemento e consumo é notablemente mellor nestas últimas.

Na segunda parte desta tese, abordouse o problema da migración de fíos de execución e páxinas de memoria en sistemas NUMA. Estes servidores contan coa particularidade de que a latencia dos accesos a memoria non é uniforme entre os distintos núcleos dos procesadores que os forman. Isto fai que aquelas operacións entre *cores* e módulos de memoria que se encontren dentro dun mesmo nodo NUMA, coñecidas como operacións locais, sexan máis rápidas que aquelas entre *cores* e memorias de distintos nodos, o que se denomina como operacións remotas. Neste contexto, a localidade dos datos é fundamental para acadar un rendemento óptimo, evitando a penalización na latencia propia dos accesos remotos. Sen embargo, é necesario acadar un equilibrio entre accesos locais e remotos, pois un exceso de operacións locais pode implicar unha saturación dos buses de memoria coa conseguinte perda de rendemento.

O actual (no momento de escritura desta tese) *scheduler* de Linux, coñecido como *Completely Fair Scheduler (CFS)*, foi un grande paso cara adiante no momento da súa incorporación pois cumpre coas expectativas de rendemento na maioría de sistemas e situacións. Nótese a complexidade desta tarefa, dada a ubicuidade de Linux a través dunha grande variedade de dispositivos: servidores, ordenadores persoais, sistemas móbiles, sistemas empotrados, etc. Como se recolle no Capítulo 3, aínda que CFS ten en conta distintos *scheduling domains*, distinguindo entre CPUs lóxicas e físicas ou incluso diferenciando os nodos NUMA do sistema, os algoritmos actualmente presentes no *kernel* seguen a estar enfocados particularmente ao reparto da carga e non tanto na mellora da localidade entre fíos de execución e datos. Co obxectivo de mellorar estas deficiencias, varios parches foron engadidos ao código do sistema operativo dos que cabe destacar a incorporación das *Transparent Huge Pages (THPs)* e do *NUMA Balancing (NB)*. O mecanismo das THPs encárgase de agrupar páxinas de memoria consecutivas e, a partir destas, crear páxinas de grande tamaño cuxa xestión é máis eficiente ca no caso das páxinas convencionais. En primeiro lugar, a frecuencia de operacións sobre o *Translation Lookaside Buffer (TLB)* é reducida, dado que cada fallo na TLB corresponde a un rango maior de direccións, polo que cada proceso ten unha maior cantidade de datos coa que traballar antes do seguinte fallo. En segunda instancia, o número de entradas na TLB vese diminuído, polo que pode ser almacenada na memoria caché de segundo nivel, mellorando a velocidade das operacións sobre o

buffer. Pola outra parte, NB almacena unha serie de estatísticas sobre cada proceso para ter en conta o histórico de accesos a memoria recentes no momento de decidir a localización dun fío de execución. NB tamén desvincula periodicamente as páxinas de memoria para movelas a outros nodos NUMA coa finalidade de mellorar a localidade das devanditas páxinas. A pesares dos esforzos realizados en mellorar CFS, pénsase que aínda queda marxe de mellora en relación á asignación e localización de fíos de execución e páxinas de memoria en sistemas NUMA.

Na busca de mellores solucións para o problema do *scheduling* en Linux, esta tese propón unha colección de algoritmos recollidos no Capítulo 3. Estas estratexias foron implementadas nunha ferramenta chamada Thanos. Dende o espazo de usuario, esta ferramenta recolle diversos datos de rendemento, procésaos e executa os algoritmos que deciden que migracións facer. O rendemento é inferido principalmente a través dos datos recollidos dos *hardware performance counters (HC)*. Estes rexistros especiais permiten avaliar distintas métricas como as instrucións executadas ou a latencia das operación de memoria realizadas, coas que se pode deducir o rendemento dunha aplicación ou do propio sistema. Unha vez que estes datos son correctamente procesados, os diferentes algoritmos deciden se é necesario realizar algunha migración, que fíos ou páxinas deben ser migradas e o seu destino.

Dentro da ferramenta distínguese entre dous tipos de estratexias: aquelas para a migración de fíos, e as deseñadas para a migración de páxinas de memoria. Ademais das métricas obtidas a partires dos contadores hardware, estes algoritmos utilizan unha series de heurísticas para tomar as decisións pertinentes ao movemento dos *threads* e das páxinas de memoria. Estas heurísticas inclúen a análise de nodos NUMA preferidos (aqueles nodos onde se realizan máis operacións de memoria), datos históricos de rendemento, ou comparativas entre fíos do mesmo proceso para detectar aqueles cuxo desempeño está considerablemente por debaixo da media.

Nesta tese recopílanse sete algoritmos para a migración de fíos: CRA, LBMA, IMAR^2, CIMAR, NIMAR, SMA e DyRMMA. *Completely Random Algorithm (CRA)* basea todas as súas decisións na aleatoriedade, sendo así un algoritmo utilizado unicamente con fins comparativos. *Lottery-Based Migration Algorithm (LBMA)* utiliza un conxunto reducido de heurísticas coas que asigna puntuacións ás potenciais migracións para, posteriormente, realizar un proceso de lotería onde aquelas migracións con maior puntuación teñen máis probabilidades de ser efectuadas. *Interchange and Migration Algorithm with performance Record and Rollback (IMAR^2)*, *Core-aware Interchange and Migration Algorithm with performance Record (CIMAR)* e *Node-aware Interchange and Migration Algorithm with performance Record (NIMAR)* pertencen a unha mesma familia de algoritmos que iterativamente expanden as funcionalidades de LBMA, considerando un maior abanico de heurísticas. Expandindo estas heurísticas, *Score Maximisation Algorithm (SMA)* busca óptimos globais, potencialmente migrando tódolos fíos en execución. De forma similar, e a partires das métricas do *3Dynamic Roofline Model (3DyRM)*, medidas cos contadores hardware, *3DyRM Migration Algorithm (DyRMMA)* busca optimizar globalmente o rendemento do sistema.

En canto aos algoritmos de migración de páxinas de memoria, tres son incluídos nesta tese: RMMA, TMMA e LMMA. *Random Memory Migration Algorithm (RMMA)* move páxinas aleatoriamente, polo que a súa finalidade é simplemente comparativa. *Threshold Memory Migration Algorithm (TMMA)* pretende migrar páxinas de memoria aos nodos preferidos considerando unicamente o número de operacións realizadas dende cada un dos nodos. *Latency Memory Migration Algorithm (LMMA)* amplía a TMMA e engade un análise da latencia para detectar e solucionar posibles problemas de conxestión.

Para avaliar as propostas recollidas nesta tese, deseñáronse tres experimentos que recollen os tres casos de uso máis habituais para os sistemas NUMA. Por unha parte, temos o *Experiment Single*, onde un único programa de proba é executado no sistema. Desta forma, o código en execución ten dispoñibles tódolos recursos do servidor. No *Experiment Interactive* recréase un entorno interactivo, a partires de datos de uso anónimos cedidos polo Centro de Supercomputación de Galicia (CESGA), no que varios usuarios lanzan pequenas tarefas en momentos arbitrarios. Así, as condicións son variables en canto a ocupación das CPUs e dispoñibilidade de recursos. No último experimento, *Experiment Queue*, tódolos *cores* do servidor mantense ocupados en todo momento emulando un sistema no que varios usuarios mandan tarefas a unha cola. Como códigos de proba utilizáronse aqueles incluídos na *suite NASA Advanced Supercomputing Parallel Benchmarks (NPB)*. Esta colección de programas son amplamente utilizados na comunidade de investigación en *high-performance computing (HPC)* e facilitan a avaliación do rendemento de sistemas, novas formas de paralelismo ou técnicas de mellora de rendemento. Nótese que dada a grande calidade de implementación destes códigos, sería de esperar pequenas ou nulas marxes de mellora no seu tempo de execución. Estes experimentos foron executados en dous servidores NUMA diferentes con distintas características para poder realizar unha análise máis completa dos algoritmos avaliados. Os resultados destes experimentos pódense atopar no Capítulo 4.

En primeiro lugar fíxose unha comparación do rendemento do actual *kernel* de Linux, CFS, coas opcións de THP e NB activadas e desactivadas. Os resultados mostran que a utilización de ambos parches é xeralmente positiva, incrementando o rendemento ata un 47 %, particularmente en situacións nas que haxa varias tarefas executándose concurrentemente, como é o caso do *Experiment Queue*. En consecuencia, considerouse como punto de referencia (*Baseline*) aos resultados extraídos con CFS con ámbolos dous parches activados.

Posteriormente, foi realizada unha comparación entre o *Baseline*, as opcións máis habituais de *mapping* como son *Direct* e *Interleave*, e os algoritmos incluídos en Thanos. No *Direct mapping*, tódalas páxinas de memoria procuran ser situadas nun mesmo nodo NUMA, sempre que a dispoñibilidade de recursos o permita. Cando se utilizan menos fíos que *cores* dentro dun nodo, estes son asignados ao mesmo nodo que as páxinas. Noutro caso, son situados ao longo doutros nodos. Así, procúrase controlar e reducir a distancia entre fíos e datos para mellorar a latencia das operacións de me-

moria. Utilizando o *Interleave*, as páxinas de memoria ubícanse a través dos nodos do sistema coma se dunha lista circular se tratase. Desta forma, maximízase o ancho de banda dispoñible a costa de pagar potenciais penalizacións na latencia. Os resultados mostran que o *Direct mapping* é unha opción moi recomendable en condicións onde haxa múltiples programas de proba en execución concorrente, grazas a que se mantén a localidade entre fíos e páxinas de memoria. Sen embargo, non sucede o mesmo cando só hai un código en execución e situar tódalas páxinas de memoria nun único nodo NUMA provoca que haxa fíos que teñan unha grande penalización na latencia de acceso aos datos, cunha perda significativa de rendemento. Por outra banda, o *Interleave* acada resultados dispares en función do sistema no que se execute e do experimento en cuestión. No *Experiment Single*, os tempos de execución vense beneficiados polo incremento no ancho de banda naqueles sistemas cunha pequena penalización de latencia nos accesos remotos. Noutros sistemas, as perdas de rendemento debidas ao aumento da latencia son demasiado grandes como para ser compensadas doutra forma. Nos experimentos onde hai máis dun código executándose concorrentemente, a perda da localidade entre fíos e datos supón un aumento do tempo de execución na maioría dos *benchmarks*.

Respecto aos algoritmos recollidos no Capítulo 3, cabe destacar os resultados acadados por tres algoritmos: *CIMAR*, *NIMAR* e *SMA*. Estes algoritmos, que soamente se encargan das decisión respecto das migracións de fíos, habitualmente conseguen mellorar os tempos conseguidos polo *Baseline* nos tres experimentos e nos dous servidores. CIMAR e NIMAR son evolucións do algoritmo IMAR^2, migrando un conxunto reducido de fíos en base un sistema de puntuación similar ao utilizado por SMA. Estes dous algoritmos distínguense principalmente pola forma de asignar a localización dos fíos. Por unha banda, CIMAR asigna os fíos a *cores* particulares, polo que se reforza o uso dos primeiros niveis da memoria caché. Sen embargo, CIMAR é subóptimo no reparto da carga computacional entre *cores*, o que pode ser un problema cando se executan concorrentemente varios códigos de altos requisitos computacionais. Pola outra, NIMAR fai unha asignación por nodo NUMA, de forma que o traballo de balanceo de carga é delegado ao sistema operativo. Desta forma pérdese lixeiramente a eficiencia no uso dos primeiros niveis de caché que se tiña con CIMAR, pero se mellora o problema do reparto de carga delegando en CFS, capaz de efectuar esta tarefa máis eficientemente dado que traballa en espazo de *kernel*. SMA utiliza un sistema de puntuación para atopar un óptimo global na localización dos fíos no sistema polo que o algoritmo pode chegar a migrar tódolos fíos nunha mesma iteración se as expectativas na mellora de rendemento son suficientemente altas.

As cargas de traballo nas que se fai un uso intensivo da memoria caché obteñen resultados lixeiramente mellores con CIMAR, mentres que en situacións con varias tarefas que se executan concorrentemente é preferible utilizar NIMAR ou SMA. Comparado co *Baseline*, acadáronse melloras nos tempos de execución de ata o 46 % no *Experiment Single*, e ata o 11 % cando se executan varios programas ao mesmo tempo.

En canto ás migracións de páxinas de memoria, só se acadaron pequenas melloras

de rendemento. Isto é debido á escaseza de información proporcionada polos HC para cada unha das páxinas de memoria. As medicións realizadas mostran que, a través dos contadores hardware, soamente se acadan 10 ou máis mostras por segundo para o 1 % das páxinas. Máis experimentos foron realizados para referendar esta teoría, onde os resultados mostran que para obter unha mellora de rendemento significativa é necesario migrar unha cantidade maior de páxinas de memoria.

Finalmente, o impacto enerxético das diferentes propostas foi avaliado. Os resultados experimentais mostran unha forte correlación entre o tempo de execución e o consumo enerxético, crecendo este último de forma linear co tempo de execución. Polo tanto, as conclusións amosadas anteriormente son tamén válidas nesta materia. Así pois, os algoritmos incluídos en Thanos conseguen mellorar o consumo enerxético, en comparación ao *Baseline*, ata un 40 % en tarefas particulares e ata un 12 % en escenarios con varios programas en execución.

Introduction

The continuous advances and innovations in computer science and technology carry with them the need to develop new algorithms, solutions and hardware to meet the demands of society. Thus, new requirements arise in terms of computational power. It is the field of high-performance computing (HPC) that is responsible for developing the necessary tools to keep increasing the performance of the computational systems.

With the end of the scaling of the Central Processing Unit (CPU) clock frequency, HPC researchers have been forced to look for new ways to improve the performance of the systems. The most common way of coping with this situation is to use parallel computing and dedicated architectures.

Despite that the concept of parallel computing made its first appearance in the late 1950s, it was not until the 1960s and 1970s that became a reality in the form of supercomputers. It was in the 1990s when parallel systems, particularly shared-memory systems, became more accessible to the general public.

Along with parallel architectures, dedicated architectures were developed to improve the performance of particular tasks. Probably, the most well-known example is the case of the Graphics Processing Unit (GPU), which was developed to accelerate the rendering of 3D graphics in the 1990s. GPUs incorporate a large number of computing units, exploiting the massively parallel nature of the graphics rendering. Thus, they can be considered a particular case of many-core architectures.

The use of general-purpose computing on GPUs (GPGPU) has grown massively in the last decade, particularly in the fields of physics simulation and machine learning. The workload is offloaded to the GPUs which, given their architectural characteristics, are particularly suited for massively parallel algorithms. However, this approach often leaves the CPUs idle, which is suboptimal in terms of resource management. Thus, it seems interesting to combine the computational power of the CPU and the GPU to achieve better performance.

With respect to memory organisation, shared-memory systems might be classified into two big groups: Uniform Memory Access (UMA) and Non-Uniform Memory Access (NUMA) systems. In the first case, all the cores of the system share the same physical memory, so the latency of the operations is approximately the same. In the second case, the cores and memory modules are grouped in nodes, in such a

way that the latency of the operations is different depending on the location of the data. Thus, those operations between cores and memory modules that are in the same NUMA node—known as local operations—are faster than those that are in different nodes—remote operations.

In NUMA systems, the placement of the threads and the data is of paramount importance to achieve a good performance. Theoretically, placing the threads and the data in the same node should reduce the latency of the memory operations. However, this is not always possible, and even if it is, the performance is not always improved due to possible contention or congestion of the memory bus. Thus, the placement of the threads and the memory pages is a complex problem and an ongoing challenge in the field of HPC.

The scheduler of the operating system (OS) is the main responsible for the placement of the threads and the data. The current—at the moment of writing this document—scheduler of Linux, known as Completely Fair Scheduler (CFS), was a big step forward since it performs well in most of the systems and scenarios. The present implementation of CFS takes into account several scheduling domains, distinguishing between physical and logical CPUs or even between NUMAs nodes. Furthermore, several patches have been included in the Linux kernel to improve the performance of this kind of system, like Transparent Huge Pages (THPs) and NUMA Balancing (NB). However, this is not enough to achieve a good performance in NUMA systems since the scheduler is primarily focused on workload balance and not so much on locality.

Within this context, the use of parallel architectures is becoming more and more popular, either by combining different kinds of processors and accelerators or even by using memory layouts with different access times. In this thesis, different solutions are proposed to take advantage of this heterogeneity to improve the performance of the applications running in them.

On one hand, this work addressed the challenge of the dynamic workload distribution in heterogeneous systems, where the CPU and the GPU are used concurrently to achieve better performance. With this objective in mind, a library named Iterative Heterogeneous Parallelism (IHP) is proposed. IHP dynamically distributes the workload between the CPU and the GPU, particularly in iterative or time-step methods. The key idea of IHP is to split the problem global domain into two subdomains that are assigned to the CPU and the GPU, respectively. Periodically, the amount of work assigned to each processor is adjusted attending to their performance looking for the best use of the computational resources.

On the other hand, aiming to improve the performance of NUMA systems, this work proposes several algorithms which are incorporated in a tool named Thanos. Working in user space, Thanos gathers performance information from several sources, processes the data and executes the algorithms that decide which is the best configuration in threads and memory pages placement and which ones should be migrated to achieve it. Performance information is mainly extracted from hardware performance counters (HC) present in modern processors. These special registers allow the user to

measure several metrics as retired instructions or the latency of memory operations. Besides the information obtained by hardware counters, these algorithms use several heuristics to make decisions about the migration of threads and memory pages. These heuristics include the analysis of preferred NUMA nodes—those nodes that store more frequently used data—, historical performance data, or comparisons between threads of the same process to detect those with performance issues. Once the data is collected and processed, the algorithms decide whether some threads or memory pages should be migrated and where they should be migrated to.

Another objective of this work is to improve energy efficiency in HPC systems. This is an important target given the growing concerns about energy consumption and the environmental impact of the HPC systems. Since energy consumption is strongly related to execution times, it would be desirable that the solutions proposed in this work improve the execution times and, consequently, the energy consumption in heterogeneous and NUMA systems.

Objectives

The main objective of this work is to develop novel solutions to improve performance in the aforementioned fields. The specific objectives are:

- To develop a library able to distribute workload between the CPU and GPU dynamically, taking advantage of heterogeneous parallelism. The library should be able to take into account the performance of the different devices to balance the amount of work assigned to each one. In the last term, the library should be able to improve the execution times of the applications that use it.

- Design and develop several algorithms to improve the performance of NUMA servers, as well as their energy efficiency. These algorithms should be able to detect performance issues and take decisions about the migration of threads and memory pages to improve the performance of the system.

- Design and develop a tool that incorporates the aforementioned algorithms and allows the user to execute them easily. This tool should extract performance information of the system from several sources, process the data and execute the algorithms that decide which threads and memory pages should be migrated.

Outline

The rest of the document is structured as follows:

- Chapter 1 describes the architectures this work is focused on and the challenges that arise when trying to maximise the performance of these architectures.

- Chapter 2 presents a brief overview of the current state of the art in GPGPU and introduces the library developed in this work to distribute the workload between the CPU and GPU dynamically, named Iterative Heterogeneous Parallelism (IHP). It also shows the results obtained with IHP and the comparison to other libraries.
- Chapter 3 describes the current Linux scheduler and how it works with NUMA systems, gives an overview of proposals in the literature and presents the tool, named Thanos, and the different algorithms developed to improve the performance of NUMA systems.
- Chapter 4 shows the results obtained with Thanos, and the aforementioned algorithms, and compare them to the current scheduler of Linux and other popular options.
- Chapter 5 explains the conclusions of this work and the future work that could be done.

Chapter 1

Parallel computing, GPUs and NUMA

Why is it when something happens, it is always you three?
— Minerva McGonagall, *Harry Potter and the Half-Blood Prince.*

This thesis has faced challenges related to high-performance computing (HPC), parallelism, and how to take the most performance of current computational architectures. Parallel computing is often related to multi-cores—also to distributed-memory clusters—, but in the last twenty years was also expanded to many-cores, Graphics Processing Units (GPUs), and Non-Uniform Memory Access (NUMA) systems. This chapter describes the platforms and architectures addressed, as well as the challenges faced in this work.

1.1 Multi-core, GPUs and NUMA systems

Academy and industry are continuously facing the problem of performance scaling in an endless path of increasing computational power. From researchers to industry professionals, they all want to run bigger programs faster.

Traditionally, the solution has been to continuously increase processor frequency. Even though, manufacturers began to hit some walls when increasing the frequency. Elemental physics state that no electrical signal can propagate at a faster speed than the speed of light, about 3×10^8 m/s in the vacuum, and about 2×10^8 m/s through copper wire. This limits either the size of the chip or the frequency at which it must work. With a target of 5 GHz, an electrical signal should not travel more than 4 cm to get from one end of the chip to the other within a clock cycle [1]. Increasing the target frequency would require a significant reduction in the size of the chips. Furthermore, frequency scaling leads to an even worse problem: heat dissipation and

power consumption. With nowadays concerns about sustainability, energy cost, and the popularisation of battery-powered devices, this becomes the main limitation.

To cope with the aforementioned problems, the computer industry decided to go for chips with several computational cores running collectively in parallel, either in distributed- or shared-memory architectures. Note that shared-memory multi-processors are also known as multi-processors or multi-core processors. This way, computational power could be increased by adding more and more cores at each generation, while keeping reasonable operating frequencies.

Formally, a multi-processor is a computer system which has two or more computing units that share access to a common memory [1]. Depending on the number of processors, it is possible to distinguish between multi-cores, with up to several dozens of cores, or many-cores with hundreds or thousands of cores. Also, multi-processors can fall into two categories depending on the architectural features: Uniform Memory Access (UMA) or Non-Uniform Memory Access (NUMA).

Another way of coping with the limitations of performance scaling is to use particular architectures that are designed to perform specific tasks. The specialisation of the architecture allows for an increase in the performance of the task at hand. Thus, accelerators can be used to perform specific tasks, such as graphics processing, signal processing, audio processing, or even general-purpose computing. Probably, the best example of accelerators is the Graphics Processing Unit (GPU). Originally designed to accelerate the rendering of 3D graphics, GPUs have been successfully adapted to perform general-purpose computing given their massive number of processors, so they can be considered a particular case of many-cores.

This work focuses on the challenges of improving performance in two kinds of systems. On one hand, it explores the possibilities of improving the performance in heterogeneous systems, where GPUs are used to accelerate the execution of a program, but also explores the potential contribution of the Central Processing Units (CPUs) in the execution. On the other hand, the challenge of improving scheduling in NUMA systems is faced, where the memory is distributed across several nodes and the location of threads and data is of paramount importance.

1.1.1 System performance

Before delving into the details of the GPUs and NUMA systems, it is necessary to understand the concept of performance. In general, it can be defined as the ability of a system to perform a given task in a given time. Depending on the needs of the user, a system with good performance is that which completes the task in the shortest time possible—focusing on execution time—, or the system that completes a collection of tasks per unit of time—focusing on throughput—[2], or even completing the task with the lowest energy consumption.

Some standard programs have been developed to measure the performance of HPC systems. These programs are also known as *benchmarks*, which are gathered into

suites. Some of the most popular benchmark suites for HPC are High Performance Conjugate Gradient (HPCG) [3], STREAM [4, 5], NASA Advanced Supercomputing Parallel Benchmarks (NPB) [6] and Standard Performance Evaluation Corporation (SPEC) [7]. These benchmarks have been used for decades to evaluate the performance of HPC systems. In general, they are based on the execution of a collection of kernels, which are the basic building blocks of the target applications. Those applications are usually written in high-level, and highly-optimisable languages, like C, C++ or Fortran. The kernels are the most time-consuming parts of the application and they are the ones that are usually measured.

In order to understand the performance of a given system or a given application, it is necessary to have a model that describes its behaviour. This model is known as *performance model* and it is used to analyse current limitations—e.g. Dynamic Random Access Memory (DRAM) bandwidth—, detect potential optimisations—e.g. single instruction, multiple data (SIMD) vectorisation—or predict the performance of a given system or application.

Over the great landscape of performance models, the Roofline Model (RM) [8] stands over the rest. This model is established as the most popular among HPC researchers due to its simplicity in data collection, analysis and display while keeping a high level of descriptiveness about the system characteristics and the performance of the target code. It should be noted that the RM is not limited to CPU-only systems, but it can be used to analyse the performance of any kind of system, including GPUs.

Two parameters are taken into account by the Roofline Model: work and intensity. Work is defined as the number of operations performed by the target program. Typically floating-point operations per second (FLOPS) have been used, but this can be extended to any kind of instructions. In the most general form, operations per second can be used. Intensity, commonly referred to as operational intensity (OI), is defined as the ratio of work per byte of DRAM traffic. The Roofline Model data is typically plotted in a two-axis logarithmic scale, where the x axis represents the OI and y shows the operations per second.

In the simplest Roofline Model plot, the maximum performance of the system is represented following the formula

$$P = \min\left(P_{\text{th}}, \beta \times I\right), \tag{1.1}$$

where the performance P is the minimum between the theoretical maximum performance of the machine, P_{th} (in GOps/s), and the product of the peak bandwidth, β (in GB/s), and the operational intensity I (in Ops/B).

The maximum theoretical performance that can be achieved in a system is known as a ceiling. Different ceilings can be used to attend to the particular characteristics of the machine, using its performance with different kinds of data types or different levels in the memory hierarchy.

Figure 1.1 shows an empty Roofline Model chart with several ceilings for a particular system. On the left side, the bandwidth limit is shown for different levels in the

memory hierarchy. Thus, the closer the data is to the CPU, the higher the bandwidth, and the higher the theoretical limit. On the right side, the limit based on the maximum FLOPS of the CPU is shown for different kinds of operations. Regular scalar operations offer the lowest performance, while vector operations offer the highest. A program located on the left-hand side is more likely to be limited by the memory bandwidth. Similarly, a program located on the right-hand side is more likely to be limited by the CPU performance.

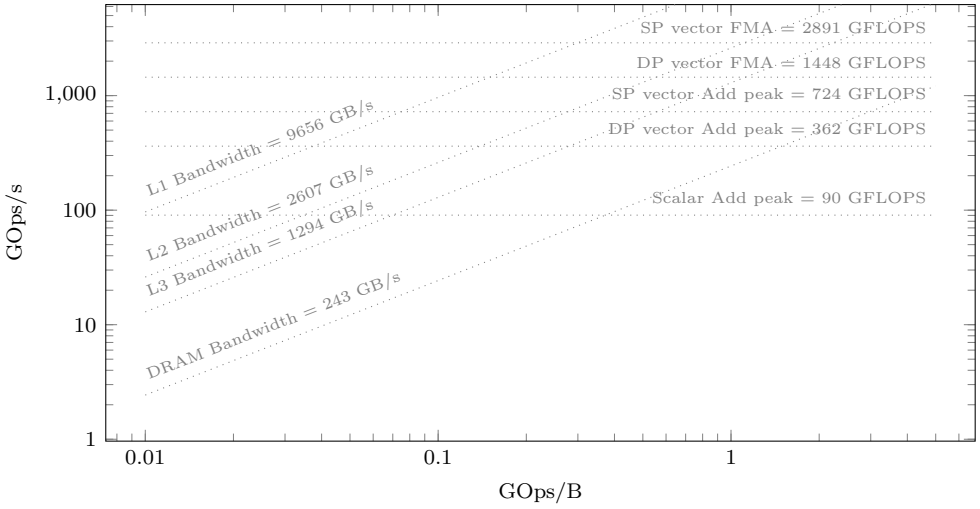


Figure 1.1: Empty roofline model chart showing different ceilings.

Figure 1.2 shows an example of the RM for several benchmarks of the NPB suite [6]. In this suite, most of the programs are under the DRAM bandwidth limit, with the exceptions of BT, FT and EP, which are above this limit, so those programs make better use of the L3 cache. Furthermore, all of them are under the CPU performance limit, so they are not taking full advantage of the vector units.

Nevertheless, the operations per second and the operational intensity are not always enough to show the performance of a code. Thus, several extensions of the Roofline Model have been proposed. For example, in [9] the RM is extended to include the ceiling of the out-of-order (OoO) execution of CPU instructions. In [10], a modification of the model is proposed to consider specific features for the GPUs, such as the High Bandwidth Memory (HBM) or the organisation of threads into warps. In the case of NUMA systems, the organisation and latency of the memory should be taken into account. For this reason, Lorenzo et al. [11, 12] introduced the 3Dynamic RM (3DyRM), adding temporal information and the average latency of memory oper-

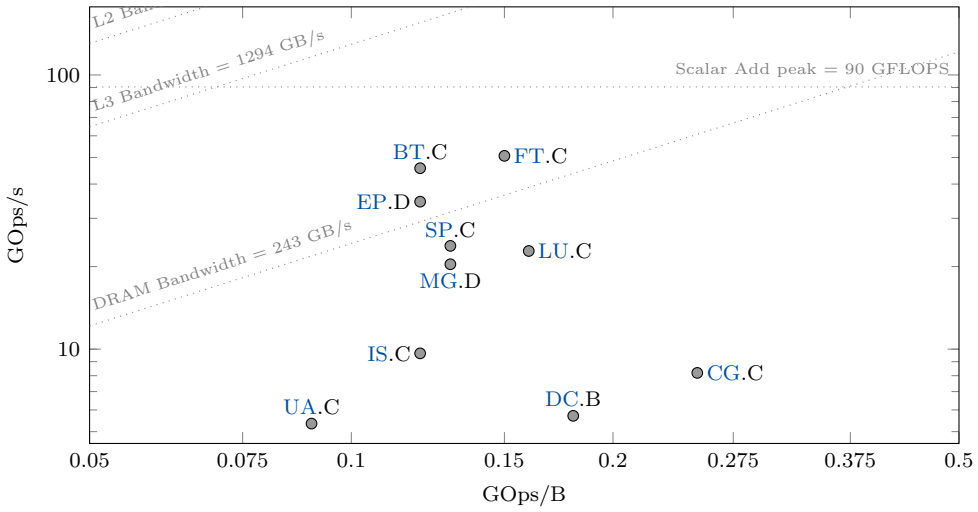


Figure 1.2: Example of roofline model chart using NASA Advanced Supercomputing Parallel Benchmarks suite.

ations to the operations per second and OI. Given that, the performance of a NUMA server can be characterised more precisely by the number of operations per second it performs, the operational intensity and the average latency of the memory operations at a given time.

1.1.2 Energy consumption in computing systems

In the last years, energy consumption has become a relevant factor in the evaluation of computing systems. This is caused by the increasing cost of energy, the popularisation of mobile and portable devices and the need to reduce the carbon footprint globally, and of the industry in particular.

Energy can be defined as the integral of the power $P(t)$ over time:

$$E = \int_{t_0}^{t} P(\tau)\,\mathrm{d}\tau\,. \tag{1.2}$$

Typically, in computer systems, power is periodically sampled and energy is approximated. Given $p_0, \ldots, p_t$ measurements of power, corresponding to times $\tau_0, \ldots, \tau_t$, energy can be approximated such that

$$E \simeq \sum_{i=1}^{t} p_i \cdot (\tau_i - \tau_{i-1})\,. \tag{1.3}$$

A processor or accelerator has two characteristic energy footprints: the peak, when it is operating at 100 % of its capacity, and the basal, when it is in idle mode. Nevertheless, the power demand of a system when not in idle mode is typically lower than the peak power.

The energy consumption depends mainly on the time the system spends in each power state. So, as a rule of thumb, the sooner the system can enter the idle state, the lower the energy consumption. With that premise, lowering execution times is one of the most promising ways to reduce energy consumption.

However, there are several ways to reduce to energetical impact of a system. Related to software, the use of more efficient instructions, algorithms, or the use of more efficient programming models can also reduce energy impact. For example, despite that vector instructions carry more power consumption than scalar instructions, they can be more energy efficient, as the performance improvement compensates for the higher power demands [13]. Data locality can also help to reduce energy consumption—for example, by improving the use of the cache, or reducing memory latency—as improving the use of cache hierarchy and reducing the number of memory accesses should reduce power demands. The use of the cache memory can be improved by the scheduler, trying to keep threads in cores whose cache memory still has valid data. The same scheduler can also help in reducing memory latency, by locating threads and memory pages as close as possible.

Related to the hardware, accelerators such as GPUs can be used to improve energy efficiency. Specific hardware that incorporates specialised units would reduce execution times, but also avoids the use of several modules that are not needed. An example of this is the use of GPUs to accelerate the execution of graphics-intensive applications, or the use of Field Programmable Gate Arrays (FPGAs) to have dedicated hardware for specific tasks.

Another way of improving energy efficiency is to use heterogeneous architectures. The Arm big.LITTLE architecture [14] is a good example of this. It combines two different types of CPUs, one with high performance and high energy consumption and another with low performance but very efficient. In this kind of system, it is the responsibility of the operating system (OS) to switch between the two types of CPUs depending on the workload [15].

1.2 GPUs and GPGPU

An important kind of architecture in the field of high-performance computing and parallel computer is the Graphics Processing Unit (GPU).

The use of specialised accelerators to process graphics data can be traced back to the 1970s when most arcade machines used that kind of hardware to improve performance and cost. Through the 1980s and 1990s, the presence of this kind of accelerator kept almost restricted to entertainment systems. It was not up to 1994

when the term GPU was first used by Sony in the context of the launch of the first PlayStation console [16], to name the 32-bit GPU, acronym at that time of *Geometry Processing Unit*. The name of GPU spread through the industry and was adopted by several manufacturers. It was in 1999 when NVIDIA used GPU as Graphics Processing Unit in the launch of the GeForce 256, with the title of "the world's first graphics processing unit (GPU)" [17].

The use of graphics accelerators kept restricted to that particular usage up to the 2000s, when several small research groups started to expand the use cases of this kind of accelerator and programmed the shaders for other purposes than graphics, so that, general-purpose computing on GPUs (GPGPU) had started. Good examples of these early works are [18] and [19], accelerating linear algebra and matrix-related computations. Since then, GPGPU is a still-growing trend for implementing algorithms which use massive parallel architectures. Two reasons lay behind this phenomenon. On one hand, GPUs provide a large number of computational cores—better known as *streaming multiprocessors* in the literature [2]. Despite that the "cores" of the GPUs are not as sophisticated as those of CPUs, they can provide great performance if the code fits the architecture. On the other hand, GPUs can be extremely efficient regarding power and energy demands.

Two technologies have prevailed as standards for GPGPU, Compute Unified Device Architecture (CUDA) [20] for NVIDIA devices and OpenCL [21] as an industry standard for general multi- and many-core processors. Nevertheless, it is CUDA that has raised as the *de facto* standard for NVIDIA GPUs due to the strong relationship between software and hardware, usually offering better performance than OpenCL [22].

It is worth mentioning that, in the last few years, a new promising technology has arisen, SYCL [23]. This "royalty-free" standard attempts to standardise GPGPU programming, provide a cross-platform solution, and make device code closer to standard C++ code, by providing a unique front-end and several implementations, known as back-ends, such as DPC++ [24] or ComputeCPP [25]. That way, a single code should work for every device supported by an SYCL back-end. At the time of writing this thesis, some back-ends support CPUs, OpenCL-compatible devices—like FPGAs—, and NVIDIA GPUs.

1.2.1 CUDA

Compute Unified Device Architecture (CUDA) is a GPGPU paradigm developed by NVIDIA, with a tight relationship between software and hardware. Thus, it is only possible to execute CUDA programs in NVIDIA GPUs[1].

[1]Some technologies exist to translate CUDA code to other languages, such as OpenCL, as a workaround for this limitation.

CUDA software

CUDA uses an extension over standard C/C++ code. *Host* code—for CPU—and *device* code—for GPU—are identified in such a way that host code is compiled by a standard C/C++ compiler like GNU Compiler Collection (GCC) [26], while device code is compiled by the NVIDIA CUDA Compiler (NVCC) [27], which is a modified version of LLVM [28].

GPU code is written in the form of *kernels*, which are noted by the keyword `__global__`. Listing 1.1 shows an example of a kernel that implements the SAXPY operation, $\vec{y} = \alpha\vec{x} + \vec{y}$, $\vec{x}, \vec{y} \in \mathbb{R}^n, \alpha \in \mathbb{R}$.

```
1 template<class T>
2 __global__ void saxpy(const T a, const T * __restrict__ x, T * __restrict__ y,
   const size_t n) {
3     const auto i = blockIdx.x * blockDim.x + threadIdx.x;
4     if (i < n) { y[i] = a * x[i] + y[i]; }
5 }
```

Listing 1.1: Example of CUDA *kernel* performing the SAXPY operation.

These kernels should be invoked in a special manner shown in Listing 1.2, by specifying the number of thread blocks and their size.

```
1 int main (const int argc, const char * argv[]) {
2     // Pre-process data to launch kernel
3     saxpy<<<numBlock, dimBlock>>>(alpha, x, y, n);
4     // Retrieve results and post-process
5 }
```

Listing 1.2: Example of invocation to *kernel* `saxpy`.

CUDA hardware

The architectural features of NVIDIA GPUs that are compatible with CUDA are very similar to the GPUs designed by other vendors like AMD, so the contents of this section can be extrapolated to other GPUs.

The most distinct characteristic of CUDA GPUs relies on thread hierarchy. GPUs are built around the concept of streaming multi-processors (SM) that handle the creation, scheduling and execution of *warps*, groups of typically 32 *CUDA threads*. When a kernel is invoked from the CPU, several blocks—a *grid*—are executed in such a way that each block comprehends several warps. Since the minimal execution unit of CUDA is the warp, all threads will execute the same code over potentially different data, exploiting single instruction, multiple thread (SIMT) parallelism—a particular case of SIMD parallelism.

For the sake of simplicity handling threads, grids and blocks can be thought of as tridimensional entities. For example, Figure 1.3 shows a grid of dimensions $3 \times 2 \times 1$ with blocks of dimensions $4 \times 3 \times 1$.

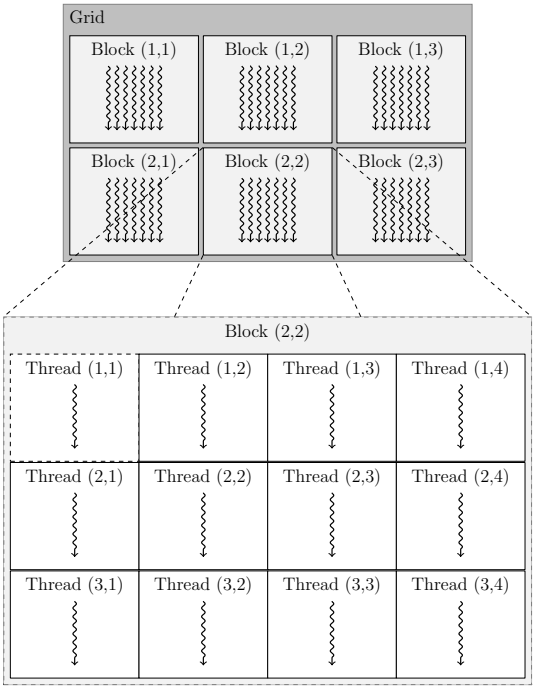


Figure 1.3: Example of thread hierarchy in CUDA.

The memory hierarchy is different in GPUs as well, with a tendency to become more similar to the CPU hierarchy. Also, compilers tend to acquire more responsibilities in this regard, making life easier for programmers. Each thread has a little local memory, similar to a set of registers. At the same time, each block has a shared memory whose content lives as much as the block itself. Finally, all blocks have access to the device global memory.

Additionally, two read-only memories exist in the GPU: constants, and textures and surfaces memories. Constants memory is designed for the storage of constant values during kernel execution with lower latency than global memory. Textures and surface memory provides different addressing forms and filters implemented in hardware, particularly focused on graphics applications.

It should be noted that recent GPUs include first- and second-level cache memories—transparent to the user—to increase performance.

1.2.2 Performance in GPGPU

The architecture of the GPUs favours their use for particular problems that are known to be "embarrassingly" parallel. These kinds of problems require little effort to be solved in parallel, or their parallel version is almost straightforward. For example, the SAXPY operation shown in Listing 1.1 is a perfect case of an embarrassingly parallel algorithm.

There are two key features required for these problems to perform well in a GPU:

- No dependencies between iterations: the result of the i-th operation should not have any influence over any j-th operation when $i \neq j$.
- Same operation applied to several data: the same transformation is used over different data to exploit SIMT parallelism.

In the case that those conditions are met, the GPUs architecture will likely be appropriated for the problem since, as a rule of thumb, a GPU is like a CPU with "a large number of dumb" cores. The number of cores in these accelerators is still increasing—the NVIDIA RTX 3070 Ti [29] has a total of 6118 cores[2]—and also their sophistication. Traditionally, GPU cores were only efficient when dealing with arithmetic operations, struggling with conditional jumps, irregular memory operations, or even double-precision instructions. Though, this is a changing tendency, where nowadays GPUs perform much better in the aforementioned cases and also incorporates dedicated cores for particular purposes like tensor operations or ray tracing computations.

Furthermore, two important phenomena should be taken into account regarding GPGPU programming: coalescent memory operations and occupancy.

As noted in [30], coalesced memory operations help to exploit the GPU memory bandwidth, thus improving performance. To achieve that, the programmer must ensure that data is correctly located in the device global memory. For example, if the n-th thread operates with the i-th element in memory, thread $n+1$ should work with the operand $i+1$. Note that GPUs architecture is evolving in such a way that can detect different patterns to perform coalesced operations, similarly to CPU prefetchers.

Occupancy is known as the relation between active warps in a SM against the theoretical maximum. As a rule of thumb, higher occupancy means better performance, since more GPU cores are active and performing computations. Though, this is not always true. Too high values can also diminish performance since available resources for each thread are reduced too [31]. Thus, it is the responsibility of the programmer to empirically ensure that optimal performance is achieved by selecting the correct grid and block dimensions.

[2] Officially: 5888 CUDA Cores, 184 Tensor Cores and 46 RT Cores.

1.2.3 Advances in GPGPU and code portability

In the last few years, several proposals focused on code portability, where the same code can be applied to several CPUs, GPUs, and other accelerators.

The traditional example of this approach is OpenCL, where a single code can be used on different devices. Despite that OpenCL favours code portability and achieved some popularity given its compatibility across AMD, NVIDIA and Intel devices, it is hard for the programmer to use in terms of development, compilation and debugging.

SYCL [23] serves as an evolution of OpenCL, whose architecture is divided into two parts, the backend and the general specification. SYCL implementations provide backends for different devices, such as CPUs, GPUs, FPGAs, etc. Each vendor can support as many devices as desired, and those can differ between implementations. The SYCL general specification gives a general Application Programming Interface (API) that all backends must follow. That ensures that the same code is supported by any backend implementation. The general specification allows C++ programmers to write SYCL kernel functions with almost every feature[3] available in the C++ standard language. Despite that some specific code is still required, the programming effort compared to bare OpenCL is significantly reduced.

Finally, NVIDIA released its HPC Software Development Kit (SDK) [32] as a suite of compilers, libraries and tools to use in NVIDIA devices in the context of HPC applications. Regarding GPGPU, it includes the CUDA compiler, `nvcc`, and a novel C++ compiler named `nvc++`, that offloads standard parallel C++ algorithms to the accelerators [33]. The latter is particularly interesting since the compiler handles all the code portability just by compiling the option `-stdpar`. The limitations of the ported GPU code are mostly the same as in SYCL, where features like exceptions or virtual functions are not supported. Nevertheless, this is probably the most promising alternative to traditional CUDA C++ dialect, where very little effort is required from the programmer if the original code makes use of standard C++ parallel execution policies.

1.3 NUMA systems

As mentioned before, shared-memory parallel systems can be classified into two main categories: UMA and NUMA systems. In conventional UMA systems, the latency of memory operations keeps a similar value no matter which CPU triggers the load/store instruction. They typically consist of a series of cores connected to a memory bus, as shown in Figure 1.4. When a memory word is referenced, its cache line is transferred from the main memory to the last level cache (LLC) of the CPU that will operate

[3] Some features such as exceptions or virtual functions are not available because of the limitations in the underlying backends.

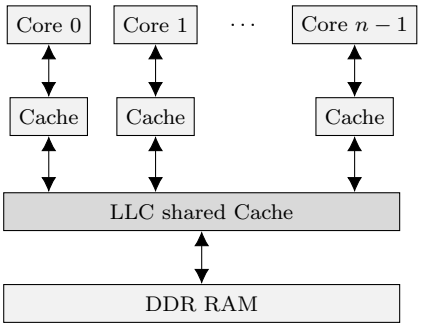


Figure 1.4: Example of a typical UMA system.

with it. To ensure the validity of every word in memory, a cache-coherence protocol must be implemented.

In contrast, NUMA systems are composed by connecting several nodes, such that each node comprehends one or more multi-processors with one or more memory slots, as illustrated in Figure 1.5. Even though the address space is shared, latencies of the memory operations depend on which CPU triggers the operation and the location of the word to operate with. Local accesses are performed if the CPU and the memory bank that stores the word are placed in the same node. If not, a remote access is performed with higher latency. Given the system shown in Figure 1.5, imagine a thread running in a core in the NUMA node 1. If the data that the thread needs is located in the same node—it performs local accesses—execution will be faster. Otherwise, data will have to travel through the interconnection network, incurring an additional latency that will slow down the execution.

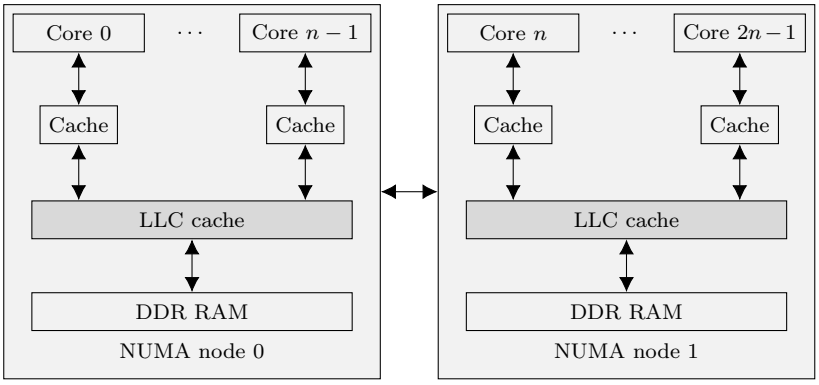


Figure 1.5: Example of a typical NUMA system.

This work focuses on a particular kind of NUMA system, called cache-coherent NUMA (cc-NUMA), and will refer to it unless another thing is indicated. cc-NUMA systems ensure that the cache memory is coherent across the nodes [1], so if a processor accesses a word in memory, it will receive the most up-to-date version of the data.

1.3.1 Challenges of NUMA systems

Despite that NUMA servers are supposed to provide greater bandwidth and better memory latencies than an equivalent UMA system, they can present performance issues and some tuning might be required.

Cache-coherent NUMAs use inter-processors communications between cache controllers to keep the integrity of the memory state when several processors access the same location. In normal conditions, the overhead of cache protocols is greatly compensated by the performance gain given by the speeds of the caches. However, cc-NUMA might perform poorly when several processors compete for the same cache lines, so false sharing takes place. This happens when processors in a shared-memory parallel system—not only in cc-NUMAs—access data placed in the same coherence block, typically a cache line, but this might extend to memory pages too [34]. In this situation, the overhead of the coherence protocols exceeds the advantages of the presence of shared memories, leading to serious performance issues.

Theoretically, local operations should be enforced to solve this problem and to take advantage of the lower latency and increased bandwidth. However, reality has shown that this is not always the best approach, and some other phenomena might take place, like memory contention, interconnection network congestion [35] or shared cache contention [36].

In the context of NUMA systems, thread and memory placement play a critical role in exploiting locality and affinity, which are known to be of paramount importance in performance.

Locality refers to the fact that applications tend to reuse the same or nearby data and instructions within a short period of time [2]. A widely known rule of thumb stands that 90 % of the time is spent in executing only 10 % of the code, highlighting the use of locality for instructions reuse. Though, data reuse is more important given the latency of DRAM memory operations, where data usually resides. As mentioned before, when accessing data in main memory, the data is moved to the significantly faster cache memory of the CPU. Thus, the latency of operations over that data is reduced until it is replaced in the cache. Two kinds of locality could be distinguished in this regard:

- Temporal locality: recently used data is likely to be accessed again.
- Spatial locality: when some data has been used, nearby data is likely to be accessed in the near future.

Programmers are encouraged to write their code in order to take advantage of locality, knowing that their application will run faster with the correct affinity, that is, threads and data are placed close to each other.

Modern OSs give the ability to bind processes or threads to a specific set of processors or CPUs. That way, those threads will be executed exclusively in those cores. Furthermore, OS developers put great effort into encouraging the default schedulers to take care of affinity, especially for NUMA.

1.3.2 Performance in NUMA systems

Several works have been presented in the literature regarding performance in NUMA systems and how affinity affects performance. A comparison included in [37] shows how performance might change for different applications using the best and worst mappings. For example, execution times might vary between 2 % up to 25 % in the NPB, depending on the sensitivity of the benchmark. Other benchmarks, like Streamcluster [38] show even higher differences. In [39], a similar comparison shows performance differences of up to 100 %, being around 15 % typically. The discrepancy in these results also arises a conclusion: not all NUMA systems are the same. NUMA servers might be significantly different in characteristics and how they are affected by locality and affinity, as shown in [40, 41]. In these works, a conventional NUMA server, with four Intel Xeon E5-4620 [42] processors, was compared with the—at that time—novel Intel Xeon Phi Knights Landing [43, 44]. Remote accesses were up to 40 % slower compared to local operations in the conventional NUMA server, but only up to 7 % in the Knights Landing.

Given the complexity of the problem and the existing differences among NUMA systems, it can be stated that this is still a non-solved topic in research that requires novel and flexible solutions.

1.4 Scheduling tasks and processes

A scheduler is a component of a system responsible for managing the execution of processes, tasks, data flows, etc. A good scheduler should ensure that the workload is balanced across the system—or systems—, the resources are used efficiently and the work is done in the best possible place. Thus, the scheduler is a critical piece of the system and one of the main responsibles for the overall performance. It is important to note that the scheduler is not only present in the OS, but also in other platforms, such as workload managers like Slurm [45] or programming models like OmpSs [46].

In workload managers, the scheduler is responsible for assigning which computing unit should execute each piece of work, and how many tasks are given to each computing resource [47]. In heterogeneous systems, the scheduler should take into account the characteristics of the computing units to assign the tasks to the most suitable one.

Other factors that should be taken into account are the cost of communications or the amount of data that needs to be transferred.

In operating systems, the scheduler is the component responsible for selecting which process—or thread—will be executed next and in which CPU [48]. It is also responsible for managing the CPU time of each process so that each process is granted a fair share of the CPU time. In a shared memory system, the scheduler should take into account several factors, like the amount of valid data in the cache, the NUMA locality, and the affinity of the processes.

This work addresses both kinds of scheduling problems. First, a library for balancing workload between CPU and GPU, named IHP, is presented in Chapter 2. Second, a user space tool named Thanos is introduced in Chapter 3. This tool incorporates several algorithms that aim to improve the performance of NUMA systems by taking into account different factors such as the NUMA locality and affinity of the processes.

Chapter 2

Heterogeneous parallelism and IHP

Avengers! Assemble.
— Steve Rogers (Captain America), *Avengers: Endgame.*

Within the context explained in Chapter 1, a brief overview of the main research in the field of heterogeneous computing is presented. Furthermore, a novel solution to address the problem of workload balancing in heterogeneous architectures—regarding collaborative CPU and GPU workload—is introduced in this chapter. The proposal developed in this work comprehends a library for iterative or time-step methods that handles automatically data movement and workload balancing among CPU and GPU.

2.1 GPGPU and heterogeneous computing

There are numerous examples in research of the successful application of GPGPU to complex problems, mostly in those where matrix operations are dominant, like numerical simulations and machine learning.

One of the first works in the topic [49] already shows important speedups when comparing GPU and CPU codes. Performance increases go up to 60× for different kernels regarding molecular dynamics, fluid dynamics or general matrix multiplication algorithms. However, these gains in performance come at a cost and programmers have to be careful when writing code for GPUs. Results in [50] show that the performance differences between an optimally tuned kernel and a regular one can be up to 17 %. This figure increases up to 235 % between a poorly programmed kernel and an optimal one.

Komatitsch et al. [51] ported to CUDA a high-order finite-element application to simulate seismic wave propagation. With the appropriate optimisation, a speed-up of 25× can be achieved compared to the CPU implementation. It should be highlighted that the application uses single-precision computations, and the authors noted that

double-precision would harm GPU performance significantly—by up to an order of magnitude.

The work of Markall et al. [52] includes a comparison between optimised CPU and GPU implementations of the matrix assembly phase of a finite element method (FEM) solver. Authors note that these implementations should use different approaches for them to execute optimally, where the GPU implementation uses an algorithm that performs more computations since it fits better its capabilities. Furthermore, they highlight that some tuning efforts should be done to achieve good performance when dealing with accelerators. In this work, a peak speed-up of almost 10× is achieved with an optimal GPU implementation over the best CPU code.

Special mention is required to what is arguably the most successful use case of GPGPU, which is its application to machine learning. Since the adoption of GPUs in this field [53], the popularity of machine learning and the quality of their results has increased dramatically. The massively parallel architecture of GPU favours the training and evaluation phases of artificial intelligence algorithms, resulting in speed-ups of up to 70×, that allow faster algorithms but also bigger models. Popular machine learning frameworks like TensorFlow [54] and PyTorch [55] make use of GPUs to accelerate their computations and are widely used nowadays.

This is only a small sample of the research done in the field of GPGPU, which is too broad to be completely covered.

Despite the big improvements in the performance of GPGPU, researchers pushed the technology by combining CPU and GPU to extract the maximum performance of their corresponding machines.

Papadrakakis et al. [56] introduced a heterogeneous approach applied to domain decomposition methods. The problem domain is decomposed into several subdomains, whose different solving stages are divided into various tasks. Using a queue of tasks, CPU and GPU kept themselves constantly busy, thus improving system-wide performance. With this methodology, the authors accomplish speed-ups of up to 50×.

Other researchers have focused on more general approaches like Maat [57], which introduces a framework for simplifying heterogeneous computing based on OpenCL kernels. Maat is supposed to handle the workload share among the different devices with various policies. Unfortunately, this project has been discontinued.

Kaleem et al. [58] use Concord [59] to offload part of the tasks to integrated GPUs. This work features two algorithms for profiling and load balancing, *naïve* and *asymmetric*, the latter being preferred by the authors. Results show that performance can be typically improved by 40 %, or up to 90 % depending on the benchmark.

LogFit [60, 61] is probably one of the most promising proposals regarding heterogeneous computing. This general-purpose library features an adaptive partitioning strategy for parallel loops, particularly focused on irregular applications. Built over Intel Thread Building Blocks (TBB) [62] and OpenCL [21], the algorithm dynamically assigns work to the GPU and the CPU. Results achieved show that performance can

be increased up to 57 %, with energy savings of up to 31 %. However, this work is particularly focused on integrated GPUs, limiting its potential.

More recent works, like [63] also combine CPU and GPU successfully using OneAPI and SYCL. This approach can be even expanded to other accelerators like FPGAs, as shown in [64].

In the author's opinion, this is still an underexplored topic in HPC research which can bring up interesting results and significant performance benefits.

2.2 Iterative Heterogeneous Parallelism (IHP)

Iterative Heterogeneous Parallelism (IHP) is a library that aims to simplify the problem of workload balancing in heterogeneous architectures, particularly for iterative or time-step problems.

The objective of IHP is to use all available computational capabilities of a given machine, thus reducing execution times, by distributing the workload among CPU and GPU. It is agnostic to the underlying technologies, so it supports multiple technologies like CUDA [20] and OpenCL [21] for GPU kernels, and OpenMP [65] or Intel TBB [62] for CPU computations. The scheme is designed to be used for iterative or time-step methods, which are very common in several domains of computer science, physics and mathematics. This kind of method follows the structure shown in Algorithm 2.1.

Algorithm 2.1 Iterative or time-step methods.

Input: Generic function $\vec{u}(\vec{x}, t)$.
Generic function $\vec{f}(\vec{u}, t)$.
Function $\vec{s}(\vec{u})$ used in the stopping criterion.
The maximum number of iterations or time-steps, $k_{\max}$.
Iterations or time-steps $t_k, k = 0, \ldots, k_{\max}$.
Output: Solution $\vec{u}(\vec{x}, t_k)$.

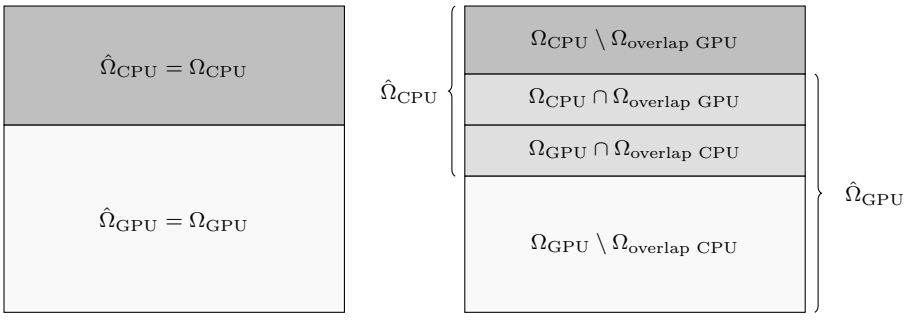
1: **for** $k = 1$ **to** $k_{\max}$ **do**
2: $\quad \vec{u}(\vec{x}, t_k) = \vec{f}(\vec{u}(\vec{x}, t_{k-1}), t_{k-1})$
3: $\quad$ **if** $\vec{s}(\vec{u}(\vec{x}, t_k))$ meets stopping criterion **then**
4: $\quad\quad$ **return** $\vec{u}(\vec{x}, t_k)$
5: **return** $\vec{u}(\vec{x}, t_k)$

IHP splits the problem domain, Ω, among the CPU and the GPU. Thus, two domains, Ω_{CPU} and Ω_{GPU}, are defined such that $\Omega = \Omega_{\text{CPU}} \cup \Omega_{\text{GPU}}$. These subdomains represent the regions for which the CPU and the GPU are responsible, respectively, and their size is ruled by a parameter $\alpha \in [0, 1]$ that represents the amount of work to be done by the CPU. For example, if $\alpha = 0.2$, Ω_{CPU} would represent the 20 % of the complete domain and Ω_{GPU} the remaining 80 % of Ω.

Since numerical methods usually use different approximations—often with higher numerical error—at the boundaries of the domain, including overlap regions help to reduce these errors at the cost of slightly higher computational demands. These overlap regions can be defined as $\Omega_{\text{overlap CPU}} \subset \Omega_{\text{GPU}}$ and $\Omega_{\text{overlap GPU}} \subset \Omega_{\text{CPU}}$. Therefore, the domains computed by CPU and GPU are defined as $\hat{\Omega}_{\text{CPU}} := \Omega_{\text{CPU}} \cup \Omega_{\text{overlap CPU}}$ and $\hat{\Omega}_{\text{GPU}} := \Omega_{\text{GPU}} \cup \Omega_{\text{overlap GPU}}$, respectively.

In the easiest case, regions are not overlapped, $\Omega_{\text{overlap CPU}} = \Omega_{\text{overlap GPU}} = \varnothing$, and the workload is distributed straightforwardly. Therefore, $\hat{\Omega}_{\text{CPU}} = \Omega_{\text{CPU}}$ and $\hat{\Omega}_{\text{GPU}} = \Omega_{\text{GPU}}$.

Figure 2.1 shows both scenarios: in Figure 2.1(a) the domain is divided into two parts without overlap, and in Figure 2.1(b) the domain is divided into two parts with overlap.



(a) Without overlap. (b) With overlap.

Figure 2.1: Representation of the domain division between CPU and GPU.

The initial value of α is estimated using a relation between theoretical peak performances of CPU and GPU, P_{CPU} and P_{GPU}, respectively, such that

$$\alpha = \frac{P_{\text{CPU}}}{P_{\text{GPU}} + P_{\text{CPU}}}. \tag{2.1}$$

This initial guess for α is expected to be not optimal, thus, a recalculation on the fly is necessary according to CPU and GPU performances in execution time. Considering that tasks are based on iterative or time-step methods, it is possible to establish a dynamic mechanism to improve the workload balance.

The key point behind IHP is to compute chunks of iterations and recompute α periodically, ensuring that CPU and GPU are used in the optimal way, that is

$$t_{\text{CPU}} \simeq t_{\text{GPU}}, \tag{2.2}$$

where t_{CPU} and t_{GPU} are the times required by the CPU and the GPU, respectively, to complete their computations for a chunk. Note that the optimal α is found in the intersection of t_{CPU} and t_{GPU}, since the iteration execution time comes from $\max(t_{\text{CPU}}, t_{\text{GPU}})$. Figure 2.2 shows a representation of a linear workload model built by IHP of CPU and GPU and the optimal value of α.

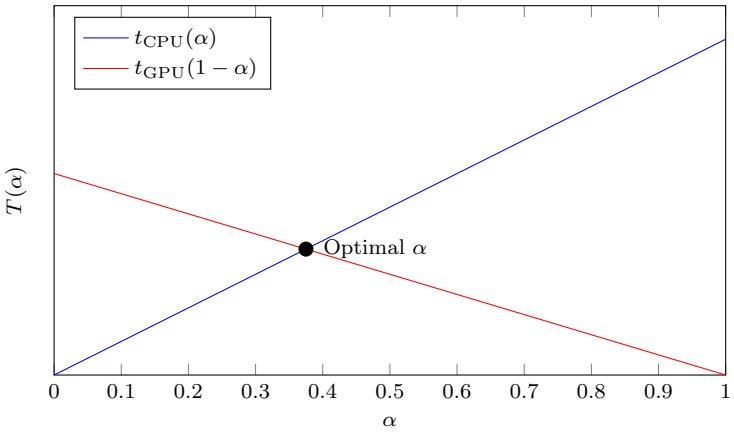


Figure 2.2: IHP linear performance model and optimal α.

Theoretically, with a stable workload through time, α converges to the optimal value after the first chunk. In the real world, two reasons justify the periodical recalculation of α. First, the workload may vary with time. For example, in simulation software, different iterations might incur different execution times due to numerical stability. Second, other processes may be assigned to the CPU or the GPU, particularly in multi-user systems, affecting performance and balance. Usually, these situations should not carry drastic changes in performance, but should not be neglected either. As a result, α is expected to slightly oscillate around the optimal value.

Note that the selection of the chunk size is also relevant in the performance of IHP. Smaller chunk sizes will allow IHP to recalculate the workload distribution more frequently, so potential workload balance issues would be solved earlier, but likely incurring more data transfers between CPU and GPU. The opposite is expected to happen with bigger chunks, so data movement should be reduced at the cost of a less reactive workload balance. Therefore, it is necessary to find a compromise between these two aspects: reactivity against performance changes or reduction of expensive data transfers.

Algorithm 2.2 shows the pseudocode of our proposal. Note that steps 6 and 7 should be computed simultaneously, so CPU and GPU should make their respective computations concurrently.

Algorithm 2.2 Heterogeneous parallelism for iterative problems.

Input: Initial value problem functions $\vec{u}(\vec{x},t)$ and $\vec{f}(\vec{u},t)$.
Function $\vec{s}(\vec{u})$ for evaluating stopping criterion.
Parameter $k_{\max}$ defining the maximum number of iterations.
Initial value for parameter α.
Minimum and maximum number of iterations per chunk, $c_{\max}$ and $c_{\min}$.
Output: Function $\vec{u}(\vec{x},t_k)$ corresponding to the result.

1: $i=0, k=1$
2: **while** $k < k_{\max}$ **do**
3: Define $\hat{\Omega}_{\text{CPU}}$ and $\hat{\Omega}_{\text{GPU}}$ such that $\text{card}(\hat{\Omega}_{\text{CPU}}) \approx \alpha\, \text{card}(\hat{\Omega}_{\text{GPU}})$
4: $c_{\text{chunk}} = \min(2^i c_{\min}, c_{\max})$
5: **for** $c=0$ **to** c_{chunk} **do**
6: $\vec{u}(\vec{x},t_{k+c}) = \vec{f}(\vec{u}(\vec{x},t_{k+c-1}),t_{k+c-1}), \quad \vec{x} \in \hat{\Omega}_{\text{CPU}}$
7: $\vec{u}(\vec{x},t_{k+c}) = \vec{f}(\vec{u}(\vec{x},t_{k+c-1}),t_{k+c-1}), \quad \vec{x} \in \hat{\Omega}_{\text{GPU}}$
8: $k = k + c_{\text{chunk}}$
9: $i = i + 1$
10: **if** $\vec{s}(\vec{u}(\vec{x},t_k))$ meets stopping criterion **then**
11: **return** $\vec{u}(\vec{x},t_k)$
12: Recompute value of α
13: **return** $\vec{u}(\vec{x},t_k)$

In order to use IHP, the user should implement the following functions:

- `alloc_dev()`: function to allocate the required memory on the device.
- `dealloc_dev()`: function to deallocate the device memory allocated in function `alloc_dev()`.
- `host_to_dev()`: code to copy data from host memory to device memory.
- `dev_to_host()`: code to copy data from device memory to host memory.
- `CPU_operator()`: code for host computations involving $\hat{\Omega}_{\text{CPU}}$.
- `GPU_operator()`: code for device computations involving $\hat{\Omega}_{\text{GPU}}$.
- `stopping_criteria()`: code for function $\vec{s}(\vec{u})$ to check if the stopping criteria is met.

Overlap is taken into account by the library itself during computation and communication related functions. However, the user has to decide how Ω_{CPU} and Ω_{GPU} are defined, as IHP intends to be a general-purpose library.

For example, in the image processing domain, the minimal working unit could be defined as a row of pixels. So, when $\alpha = 0.2$, the CPU will be responsible for the calculations of the 20 % of rows of the image and the defined overlap rows.

Source code for IHP and simple example codes can be found in [66].

2.2.1 IHPv1: assuming linear workloads

In its first version, as published in [67] and [68], IHP works under the assumption that workload is linear. This allows for very quick and efficient computation of the workload share, following the expression

$$\alpha = \frac{t_{\text{GPU}}}{t_{\text{CPU}} + t_{\text{GPU}}}, \tag{2.3}$$

where t_{CPU} and t_{GPU} represent the amount of time required by the CPU and the GPU, respectively, to complete their computations for the last chunk. The main flaw of this approach is that α might be computed inaccurately for workloads that are not linear.

2.2.2 IHPv2: considering non-linear workloads

To address the aforementioned flaw of IHP version 1 (IHPv1) regarding the assumption of linear workload, IHP version 2 (IHPv2) considers different kinds of models. For that, historical data is stored and processed to find the model that fits best for both CPU and GPU, and computes the optimal workload share according to these models. The historical data also helps with the stability of α, potentially saving time in data movement across CPU and GPU memories.

CPU and GPU performance models

Three main models can be expected in terms of performance: linear, logarithmic and exponential. There are typical examples for all of them. The complexity of most vector operations grows linearly with the size of the vector. Searches in trees and sorted groups of data typically show a logarithmic behaviour. Matrix operations and other complex linear algebra operations have an execution time that increases exponentially with the size of the matrices. Furthermore, it is possible to consider that algorithms which show other kinds of behaviours can be approximated by those three models.

Note that two models have to be built, one for the CPU and another for the GPU. Architectural differences between the two processors might lead to different behaviours and it is also possible that both models are not of the same kind. For example, the implementation of the host code might show a linear behaviour, while the device code might be logarithmic.

Fitting models

IHP uses information obtained on-the-fly, regarding how much work each kind of processor did and how long it took. The last n samples are stored, each sample containing the tuple $(\alpha, t_{\text{CPU}}, t_{\text{GPU}})$. Those values are averaged and stored in the tuple $\left(\bar{A}, \bar{T}_{\text{CPU}}, \bar{T}_{\text{GPU}}\right)$, which are later used to fit the linear, logarithmic and exponential models.

Different ways of computing the averages, imply differences in the behaviour of the balancing algorithm. Some of the most common weighted averaging methods are the following:

- The arithmetic mean—or average—is the simplest averaging method, both conceptually and in terms of implementation, see equation (2.4). As a positive, it provides more consistency over time. On the negative, it is less responsive to abrupt changes. Note that this method weights the same for every sample, regardless of the accuracy of that balance or how recent it is.

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i. \tag{2.4}$$

- Linearly Weighted Moving Average (LWMA), see equation (2.5), makes the more recent measurements to be more relevant in a linear way. This makes the algorithm balance faster in case of important changes in performance, while still considering historical data.

$$\bar{x} = \frac{\sum_{i=1}^{n} i x_i}{\sum_{i=1}^{n} i} \tag{2.5}$$

- Exponentially Weighted Moving Average (EWMA), see equation (2.6) follows the same idea of LWMA, with the difference that weight changes exponentially. Thus, more recent samples have much higher weight than the older ones.

$$\bar{x} = d \cdot \sum_{i=1}^{n} (1-d)^{n-i+1} x_i, \tag{2.6}$$

  where $d \in (0, 1)$ is the decay factor, which controls the rate of exponential decay.

- Weighting based on balance gives more relevance to those samples which implied better workload balance. For each sample, when workload share is recomputed, it is assigned a weight following

$$w_i = \exp\left(-\frac{|t_{\text{CPU}} - t_{\text{GPU}}|}{\max\left(t_{\text{CPU}}, t_{\text{GPU}}\right)}\right). \tag{2.7}$$

So that, the average is computed with

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i w_i}{\sum_{i=1}^{n} w_i}. \tag{2.8}$$

Those methods are illustrated in Figure 2.3.

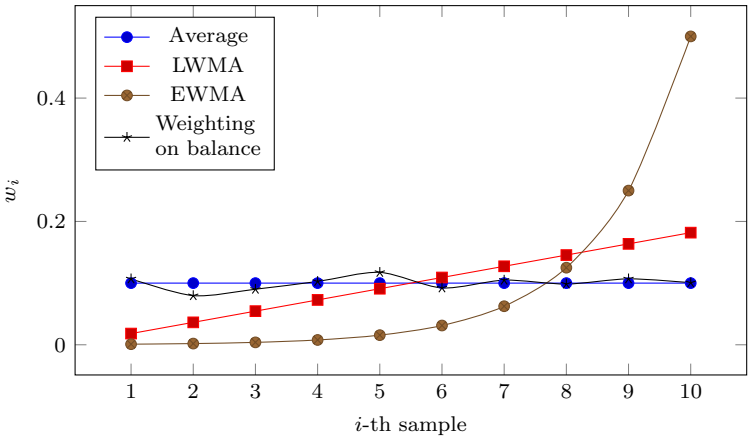


Figure 2.3: Representation of weighting methods and assigned weights when considering the last $n = 10$ measurements.

Workload models

With the tuple $\left(\bar{A}, \bar{T}_{\text{CPU}}, \bar{T}_{\text{GPU}}\right)$, the linear, logarithmic and exponential models can be built. There is a constraint in these models, they should go through $(0, 0)$, since no workload should carry zero execution time. When building the model for the CPU, $\bar{x} = \bar{A}, \bar{y} = \bar{T}_{\text{CPU}}$, and $\bar{x} = 1 - \bar{A}, \bar{y} = \bar{T}_{\text{GPU}}$ for the GPU.

The considered models are the following:

- Linear model: this is the simplest model in both terms of theoretical and practical simplicity. The execution time is approximated with a straight line that goes from $(0, 0)$ to some calculated $(\bar{x}, \bar{y})$. Thus, the execution times are expected to follow

$$f_{\text{lin}}(\alpha) = b\alpha, \quad b = \frac{\bar{y}}{\bar{x}}. \tag{2.9}$$

- Logarithmic model: for fitting the logarithmic model, a shift in the x axis is required, so that

$$f_{\log}(\alpha) = b \cdot \log(\alpha + 1), \quad b = \frac{\bar{y}}{\log(\bar{x} + 1)}. \tag{2.10}$$

- Exponential model: in the case of the exponential model, the shift has to be done in the y axis. Therefore,

$$f_{\exp}(\alpha) = \exp(b\alpha) - 1, \quad b = \frac{\log(\bar{y}+1)}{\bar{x}}. \tag{2.11}$$

Trade-offs

By increasing the number of samples taken into consideration, n, the balancing method gets less responsive to changes in the performance of the CPU or GPU. On one hand, it will provide better stability under normal system noise, potentially reducing data movement due to the changes in workload balancing. On the other hand, responsiveness is reduced in situations where a big change in workload balance is required. Imagine, for example, a scenario where CPU receives a large number of tasks from other processes, thus reducing its performance in the process handled by IHP. In the extreme case with $n = 1$, the library will assign much more work to the GPU immediately. As n increases, the time it takes to find the new optimal balance increases too. Figure 2.4 shows how α would change in a situation like this.

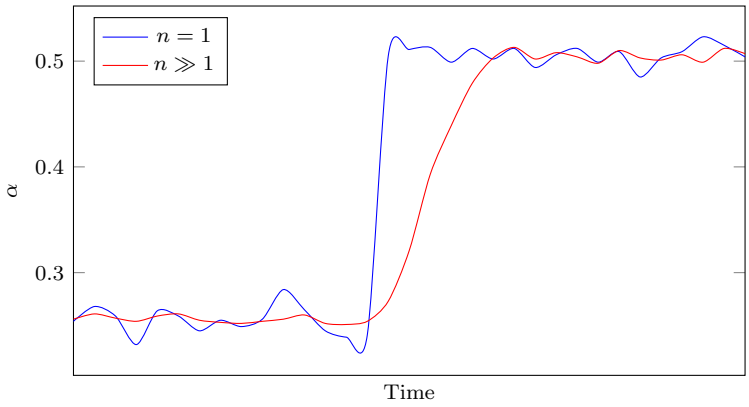


Figure 2.4: Example of behaviour under a large change of performance.

Another drawback of selecting high values for n resides in the computation time taken to compute the weighted average. Nevertheless, this overhead should be negligible for small values of n, from 5 to 15, for example.

The weighting average method also changes the behaviour of the algorithm, as discussed previously.

Selecting a model

To select which model fits the actual workload, the traditional Mean Squared Error (MSE) can be used. Let $A = \{\alpha_1, \ldots, \alpha_n\}$ be the n last values of α used, $T = \{t_1, \ldots, t_n\}$ the corresponding execution times, and $f(\alpha)$ the function that fits the performance data, with n samples each, the MSE is computed following

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (t_i - f(\alpha_i))^2 . \tag{2.12}$$

Furthermore, the Weighted Mean Squared Error (WMSE) can be used,

$$\text{WMSE} = \frac{1}{n} \frac{\sum_{i=1}^{n} (t_i - f(\alpha_i))^2}{\sum_{i=1}^{n} w_i}, \tag{2.13}$$

with $W = \{w_1, \ldots, w_n\}$ being the collection of weights for the data in T.

For both CPU and GPU, the model which has the lowest WMSE error is selected.

Recalculating workload share

Let f_{CPU} and f_{GPU} be the functions that fit the CPU and GPU workloads the best, as explained previously.

To determine the new value of α, the following problem has to be solved:

Problem 2.1. *Given f_{CPU} and f_{GPU}, find α such that:*

$$f_{CPU}(\alpha) - f_{GPU}(1 - \alpha) = 0. \tag{2.14}$$

If the performance difference between the CPU and GPU is too large, the solution to Problem 2.1 could be $\alpha \simeq 0$ or $\alpha \simeq 1$. In that cases where $\alpha < 0.01$ or $\alpha > 0.99$, the GPU or CPU will be assigned all the work, respectively, since it is considered that the contribution of the other device does not compensate for the overhead of data movement.

To solve Problem 2.1, the Newton method is used:

Step 0: Let $i = 0$, α_0 the last used value of α and $\delta \ll 1$ a threshold for accepting a solution.

Step 1: For the i-th step, compute

$$\alpha_{i+1} = \alpha_i - \frac{f_{\text{CPU}}(\alpha_i) - f_{\text{GPU}}(1 - \alpha_i)}{f'_{\text{CPU}}(\alpha_i) - f'_{\text{GPU}}(1 - \alpha_i)}, \tag{2.15}$$

where f'_{CPU} and f'_{GPU} are the derivatives of f_{CPU} and f_{GPU}, respectively.

Step 2: If $|f_{\text{CPU}}(\alpha_i) - f_{\text{GPU}}(1 - \alpha_i)| < \delta$, accept the last solution. Otherwise, $i = i + 1$, and go to Step 1.

2.3 Image denoising as case study

This section introduces the mathematical problem to be solved in the context of image denoising. In particular, diffusion methods, like the ones proposed in [69], are used as they provide an easy-to-understand example and a good case study and validation scenario for IHP.

A digital image can be described as a function $\vec{I}(x,y) : \mathbb{R}^2 \to \mathbb{R}^c$, where c corresponds to the number of channels. Typically, $c = 3$ for the common RGB—Red Green Blue—colour representation. The image domain $\Omega = (0, L_x) \times (0, L_y) \subset \mathbb{R}^2$ is defined, where L_x and L_y represent the dimensions of the image in the x and y axis, respectively.

Diffusion methods proposed in [69] are a set of coupled partial differential equations (PDEs), where an image is considered the combination of brightness—magnitude—and chromaticity—direction. So, in an instant t, an image is defined as $\vec{I}(x,y,t) = \mathbb{R}^2 \times [0,\tau) \to \mathbb{R}^c$, where $(x,y) \in \Omega$ are the spatial coordinates and t represents time. Therefore, any image can be defined as the combination of its magnitude, $M(x,y,t) : \mathbb{R}^2 \times [0,\tau) \to \mathbb{R}$, and its unit direction, $\vec{D}(x,y,t) : \mathbb{R}^2 \times [0,\tau) \to \mathbb{R}^c$. These functions are written as

$$M(x,y,t) = \left(\sum_{i=1}^{c} I_i(x,y,t)^2\right)^{\frac{1}{2}} = \left\|\vec{I}(x,y,t)\right\|, \tag{2.16}$$

$$\vec{D}(x,y,t) = \frac{1}{M(x,y,t)}\vec{I}(x,y,t). \tag{2.17}$$

To eliminate noise, diffusion flows are applied to brightness and chromaticity, respectively, so the image evolves through time.

2.3.1 Brightness diffusion flows

For the brightness, two types of diffusion flows are considered. First, the common Laplacian flow, based on the heat equation,

$$\frac{\partial M}{\partial t}(x,y,t) = M_{xx}(x,y,t) + M_{yy}(x,y,t) = \Delta M(x,y,t), \tag{2.18}$$

where the subscripts denote partial derivatives. As this is an isotropic flow, the behaviour is the same in every direction, and as a consequence, flaws in preserving the edges of the objects of the image. Second, the more refined anisotropic flow [69], that is introduced as

$$\frac{\partial M}{\partial t}(x,y,t) = \frac{\left(M_{xx}M_y^2 - 2M_xM_yM_{xy} + M_{yy}M_x^2\right)^{\frac{1}{3}}}{1 + \|\nabla M\|}, \tag{2.19}$$

where the anisotropic behaviour is introduced by the term $1/(1+\|\nabla M\|)$. In the edges of objects in the image, the norm of the gradient grows up, preventing diffusion in these areas.

2.3.2 Chromaticity diffusion

The chromaticity is defined in (2.17) and is represented by a unit vector field that defines the direction of the colour for a given pixel. The i-th component of $\vec{D}(x,y,t)$ is denoted as $D_i(x,y,t) : \mathbb{R}^2 \times [0,\tau) \to \mathbb{R}$, with $i = 1,\dots,c$, indicating how much that vector is drawn to a specific colour—red, green or blue in RGB codification.

Diffusion flows for chromaticity come from solutions of Problem 2.2.

Problem 2.2. *Find $\vec{D}(x,y,t) : \mathbb{R}^3 \to \mathbb{R}^c$, such that minimises:*

$$\min_{\vec{D}:\mathbb{R}^3\to\mathbb{R}^c} \int_\Omega \left\|\nabla \vec{D}(x,y,t)\right\|^p \mathrm{d}x\,\mathrm{d}y\,, \quad t \in [0,\tau), p \geq 1, \tag{2.20}$$

subject to $\left\|\vec{D}(x,y,t)\right\| = 1, \quad \forall (x,y) \in \Omega, t \in [0,\tau).$

It can be proved that the gradient descent flow of (2.20) has the form

$$\frac{\partial D_i}{\partial t} = \operatorname{div}\left(\left\|\nabla \vec{D}\right\|^{p-2} \nabla D_i\right) + D_i \left\|\nabla \vec{D}\right\|^p, \quad 1 \leq i \leq c. \tag{2.21}$$

To narrow down the scope of this work, the two more interesting values of p are considered. The first one is $p = 2$, in which the isotropic diffusion flow is obtained by

$$\frac{\partial D_i}{\partial t} = \Delta D_i + D_i \left\|\nabla \vec{D}\right\|^2, \quad 1 \leq i \leq c. \tag{2.22}$$

The second value is $p = 1$, obtaining the anisotropic flow for the chromaticity:

$$\frac{\partial D_i}{\partial t} = \operatorname{div}\left(\left\|\nabla \vec{D}\right\|^{-1} \nabla D_i\right) + D_i \left\|\nabla \vec{D}\right\|, \quad 1 \leq i \leq c. \tag{2.23}$$

2.3.3 Discretisation

Attending to digital images structure, a uniform mesh with $N+2$ points in the x axis and $M+2$ in the y axis is proposed, so the distance between two points in the mesh for each axis is $h = h_x = L_x/(N+1) = h_y = L_y/(M+1) = 1$. This mesh is denoted as Ω_h and it is defined as the set of nodes

$$\Omega_h = \{(x_i, y_i)\,, 1 \leq i \leq N, 1 \leq j \leq M\}\,, \tag{2.24}$$

and its boundary domain is

$$\partial\Omega_h = \{(0, y_j), (L_x, y_j), 0 \leq j \leq M+1\} \cup \{(x_i, 0), (x_i, L_y), 0 \leq i \leq N+1\}. \quad (2.25)$$

Therefore, domain Ω is represented by the set of nodes $(x_i, y_j) = (ih_x, jh_y)$, $i = 0, \ldots, N+1$, $j = 0, \ldots, M+1$.

Additionally, a discretisation of time is required in these kinds of time-step methods, so a time step δ is defined, such that k-th instant is $t_k = k\delta$ for $k = 0, \ldots, k_{\max}$.

Finally, in this section, the following notation is used

$$(x_i, y_j, t_k) = (ih_x, jh_y, k\delta), \quad i = 1, \ldots, N, j = 1, \ldots, M, k = 1, \ldots, k_{\max}, \quad (2.26)$$

$$\vec{u}(x_i, y_j, t_k) \simeq \vec{u}_{i,j,k}, \quad i = 1, \ldots, N, j = 1, \ldots, M, k = 1, \ldots, k_{\max}, \quad (2.27)$$

for any given function $\vec{u}(x, y, t)$.

Concerning initial conditions, $u(x, y, 0) = M(x, y, 0)$ and $\vec{u}(x, y, 0) = \vec{D}(x, y, 0)$ for brightness and chromaticity, respectively. For boundary conditions $\vec{u}(x, y, 0) = \vec{0}$ for every $(x, y) \in \partial\Omega_h$.

2.3.4 Numerical solution

For computing the numerical solution of the PDEs (2.18), (2.19), (2.22), and (2.23), finite difference schemes are used for the spatial derivatives and an Explicit Euler method for temporal derivatives. Centred differences are used whenever possible for first-order spatial derivatives. Note that, in the points (x_i, y_j) with $i = \{1, N\}$ or $j = \{1, M\}$, a different approximation is required, by using forward differences when $i = 1$ or $j = 1$ and backward differences when $i = N$ or $j = M$. Second-order spatial derivatives are computed using centred differences in all cases, considering that $\vec{u}_{i,j,k} = 0$ if $(x_i, y_j) \notin \Omega_h$.

An explicit Euler scheme is used for temporal derivatives,

$$\begin{cases} \vec{u}(x, y, t_0) = \vec{u}(x, y, 0), \\ \vec{u}(x, y, t_k) = \vec{u}(x, y, t_{k-1}) + \delta \vec{f}(\vec{u}(x, y, t_{k-1}), t_{k-1}), \quad k = 1, \ldots, k_{\max}, \end{cases} \quad (2.28)$$

where, $\vec{u}(x, y, t)$ denotes brightness or chromaticity function and $\vec{f}(\vec{u}, t)$ represents the corresponding isotropic or anisotropic flow function.

Additionally, the integral in (2.20) provides an estimation of the amount of noise of an image, considering it as an energy measure. Thus, the following function is defined:

$$E(\vec{u}) = \int_\Omega \|\nabla \vec{u}(x, y, t)\|^p \, \mathrm{d}x \, \mathrm{d}y, \quad (2.29)$$

hereinafter referred to as the image energy. Since noise alters the values of the pixels, the norm of the gradient increases and, consequently, the energy. In this work, the

general value of $p = 2$ has been used in (2.29), preserving the case with $p = 1$ for the anisotropic flow for chromaticity.

Computing (2.29) involves a discretisation, so the integral of $\|\nabla \vec{u}(x, y, t)\|^p$ is approximated in each rectangle of Ω_h considering the trapezium rule in each iterated integral. Therefore, (2.29) is approximated as

$$E(\vec{u}) = \int_\Omega \|\nabla \vec{u}\|^p \, \mathrm{d}x \, \mathrm{d}y \simeq$$
$$\frac{h^2}{4} \sum_{i=1}^{N-1} \sum_{j=1}^{M-1} \left(\|\nabla \vec{u}_{i,j,k}\|^p + \|\nabla \vec{u}_{i+1,j,k}\|^p + \|\nabla \vec{u}_{i,j+1,k}\|^p + \|\nabla \vec{u}_{i+1,j+1,k}\|^p \right). \quad (2.30)$$

The values of the energy of the image are used as a stopping criterion, so the diffusion flow stops once the following condition is fulfilled

$$\frac{E(\vec{u}(x, y, t_k))}{E(\vec{u}(x, y, t_0))} \leq \rho, \quad k \in [0, k_{\max}], \rho \in (0, 1). \quad (2.31)$$

Therefore, the final time τ is t_k for the first t_k that fulfils this condition. As the energy value is only used as a guide to stop the method, maximum precision is not needed, while a low computation time is preferable. Additionally, in order to save computation time, it is not necessary to compute the energy in every iteration, as it holds

$$\lim_{t \to \infty} E(\vec{u}(x, y, t)) = 0, \quad (2.32)$$

decreasing fast when t is close to 0, but slowing down as t increases as shown in Figure 2.5.

2.3.5 Definitions of IHP domains

As mentioned in Section 2.2, IHP splits the domain Ω_h into several subdomains in order to balance the workload between CPU and GPU. In the case of image denoising, the split of the domain is almost straightforward, as it can be divided by rows. Therefore, the subdomains of Ω_h are defined like

$$\hat{\Omega}_{\text{CPU}} = \{(x_i, y_j), 1 \leq i \leq N, 1 \leq j \leq r_{\text{split}}\}, \quad (2.33)$$
$$\Omega_{\text{overlap CPU}} = \{(x_i, y_j), 1 \leq i \leq N, r_{\text{split}} < j \leq r_{\text{split}} + r_{\text{overlap}}\}, \quad (2.34)$$
$$\hat{\Omega}_{\text{GPU}} = \{(x_i, y_j), 1 \leq i \leq N, r_{\text{split}} < j \leq M\}, \quad (2.35)$$
$$\Omega_{\text{overlap GPU}} = \{(x_i, y_j), 1 \leq i \leq N, r_{\text{split}} - r_{\text{overlap}} \leq j \leq r_{\text{split}}\}, \quad (2.36)$$

where $r_{\text{split}} = \lfloor \alpha M \rfloor$ denotes the row where the domain is split, and r_{overlap} denotes the number of rows that define the overlap region.

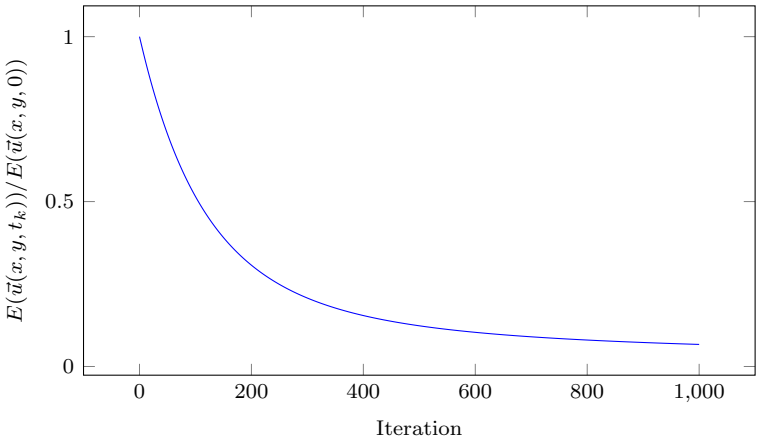


Figure 2.5: Example of the behaviour of energy function in the denoising of the brightness of an image.

Concerning the overlap region, the anisotropic flow for chromaticity is the diffusion flow that needs more points in the calculations. Computing the divergence of the points in the i-th row requires the gradients of the points in the rows $i-1$ and $i+1$. These gradients are computed with the values of the points in the rows $i-2$, i, and i, $i+2$ respectively. Given that, $r_{\text{overlap}} = 2$.

2.4 Experimental environment

To validate the proposed methodology, several tests have been carried out in two systems with different characteristics regarding computational capabilities:

- Desktop: a Linux system, kernel version 5.15.0 with an Intel Core i5-7600 processor [70], with 4 cores—4 threads—, Kaby Lake architecture, 6 MB L3 cache, 3.50 GHz–4.10 GHz, an NVIDIA GTX 1050 Ti GPU [71] with 768 NVIDIA CUDA cores, 1.29 GHz–1.39 GHz, and 32 GB of DDR3 memory. Two implementations were used in this system. On one hand, a CPU-only implementation using Intel TBB, with a GPU-only version using OpenCL, and a IHP implementation using both Intel TBB and OpenCL. The compiler GCC 7.4.0 [26] was used with the highest optimisation options enabled. On the other hand, a CPU-only implementation using OpenMP [65], with a GPU-only version using CUDA [72], and the IHP implementations use both OpenMP and CUDA. Compilers GCC 10.3.0 [26] and NVCC v11.6.112 [27] were used with the highest optimisation options enabled.

- Laptop: a Linux system, kernel version, CPU i7-5700HQ [73] with 4 cores—8 threads—, Broadwell architecture, 6 MB L3 cache, 2.70 GHz–3.50 GHz, an integrated GPU Intel Iris Pro 6200[1], 0.30 GHz–1.05 GHz, and 16 GB of DDR3 memory. In this system, the CPU-only implementation uses Intel TBB, the GPU-only version uses OpenCL, and the IHP implementations use both Intel TBB and OpenCL. The compiler GCC 9.4.0 [26] was used with the highest optimisation options enabled.

According to (2.1), initially, $\alpha = 0.03$ and $\alpha = 0.5$ for systems Desktop and Laptop, respectively.

Results in this section come from the average of 5 independent executions of the image denoising algorithms with an image of 4000×6016 pixels.

It should be noted that the main difference between the isotropic and anisotropic diffusion flows is that the latter involve more arithmetic operations. Therefore, the anisotropic flows are more computationally intensive than the isotropic ones.

2.5 Results with different kinds of workloads

IHPv1 and IHPv2 have been tested also under linear, logarithmic and exponential workloads. To emulate those, the computations for each pixel of the images are repeated several times. In the linear case, each pixel value is computed 10 times. For the logarithmic, computations are repeated $\lceil 10 - 5\alpha \rceil$ times for each pixel. Thus, the higher value of α, the least time is spent in the kernel. Finally, in the exponential workload, each computation is repeated $\lceil 10 + 20\alpha \rceil$ times. These modifications allow obtaining the behaviours shown in Figure 2.6.

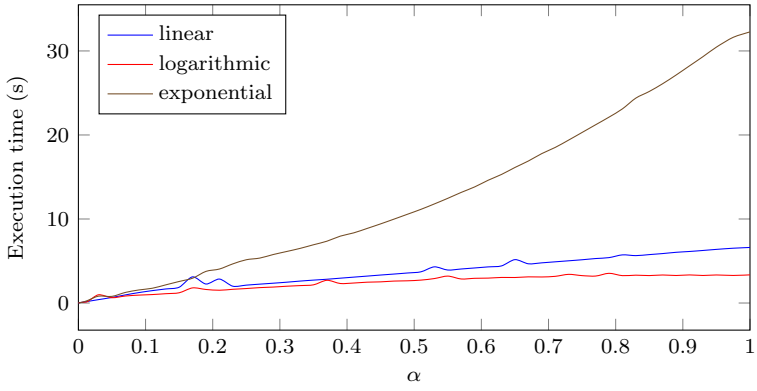


Figure 2.6: Execution time evolution for linear, logarithmic and exponential workloads.

[1]The OpenCL controller for this GPU only supported single-precision computations.

Note that the exponential workload has the biggest potential of improving execution times when using heterogeneous parallelism since extracting part of the work from one device reduces drastically the amount of total work to be done. It is the opposite for the logarithmic workload, though.

Concerning diffusion methods parameters, 200 and 50 iterations have been computed in systems Desktop and Laptop, respectively, with chunks $c_{\min} = 1$ and $c_{\max} = 10$ for both brightness and chromaticity.

Results show that significant improvements can be obtained, in terms of execution time, when combining CPU and GPU computing power compared to GPU-only implementations. This effect can be observed in the three tested workloads, particularly in the exponential workload, and the two systems.

As shown in Figures 2.7 and 2.8, speed-ups of up to $\times 1.61$ and $\times 2.60$ can be achieved for Desktop and Laptop systems, respectively. Detailed results are shown in Tables A.1 to A.9 in Appendix A.

In the case of the Desktop system, there is a large gap between CPU and GPU performance. When using single-precision instructions, the real performance of the CPU is about 10 % of the GPU. Despite this difference, it is still worth combining both kinds of processors in terms of execution time.

For the linear kind of workloads, the CPU typically handle around the 6 % of the computations, as shown in Figure 2.9, resulting in an improvement of execution times around the 4 %.

For logarithmic workloads, the CPU contribution is much smaller and even discarded as is the case in the anisotropic flow for brightness processing. Two main reasons lay behind this phenomenon. First, the CUDA implementation already produces low execution times, giving little room for improvement. Second, during the time that IHP invests searching for optimal workload shares, some time is lost due to imbalance, becoming a dominant factor in the overhead. This is not a big issue, though, since the time loss is in the order of tenths of a second, which might be considered negligible in this scenario. Despite that, the loss in performance is less than 1 %.

The exponential workloads are the most benefited, where IHP version increases performance approximately by 30 %. Note that, in this case, dividing the work between CPU and GPU reduces the workload overall, being easier to improve the execution times. When using double-precision registers, the gap between CPU and GPU performance is reduced and the benefits of using heterogeneous parallelism are greater. As a result, the achieved speed-up grows up to $\times 1.60$.

In the Laptop system, CPU and GPU performances are closer. This allows the CPU to handle larger parts of the workload, so the heterogeneous implementations are more effective, see Figure 2.8. For linear workloads, execution times improve between 36 % and 39 %. In the case of logarithmic workloads, the improvements vary between 5 % and 43 %. It is the exponential workloads where execution times are furthest reduced, with speed-ups of up to $\times 2.60$.

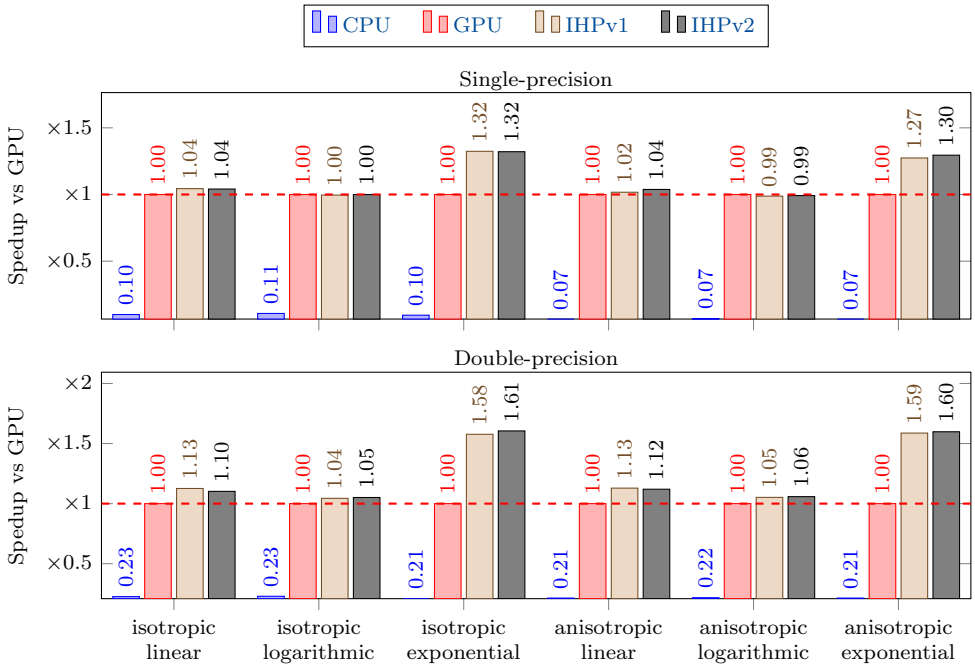


Figure 2.7: Speed-up against GPU for different flows for brightness and chromaticity and various workload types on the Desktop system. Higher is better.

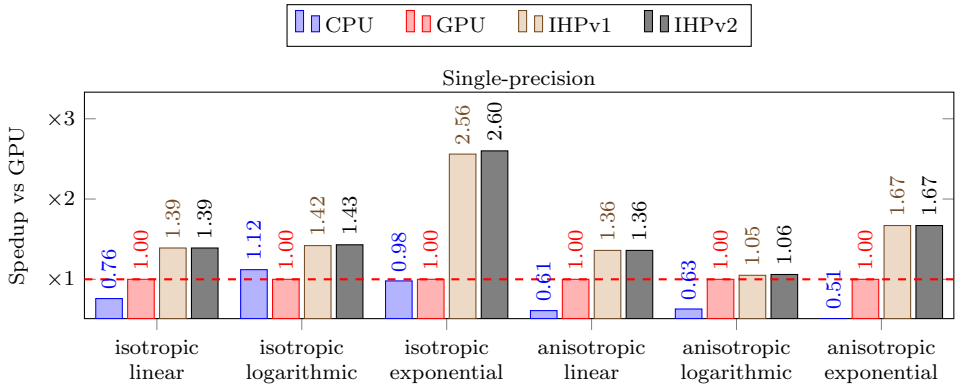


Figure 2.8: Speed-up against GPU for different flows for brightness and chromaticity and various workload types on the Laptop system. Single-precision computations. Higher is better.

Finally, the differences between IHPv1 and IHPv2 should be discussed. As mentioned in Section 2.5.1, the region around the optimal workload share is linear. That means that if IHPv1 reaches that point, the workload it computes will be near the optimal, and differences with IHPv2 are negligible in that regard. Nevertheless, the storage of historical data and the new mechanisms introduced in IHPv2 allows for more stable values of α that reduce the data movement between CPU and GPU. Figures 2.9 and 2.10 show examples of how the workload share can change through the execution in the Desktop system. It is possible to see several oscillations of α when using IHPv1 that do not happen with IHPv2. These oscillations carry data movements between CPU and GPU that increase the overhead, slightly hurting execution times in the process.

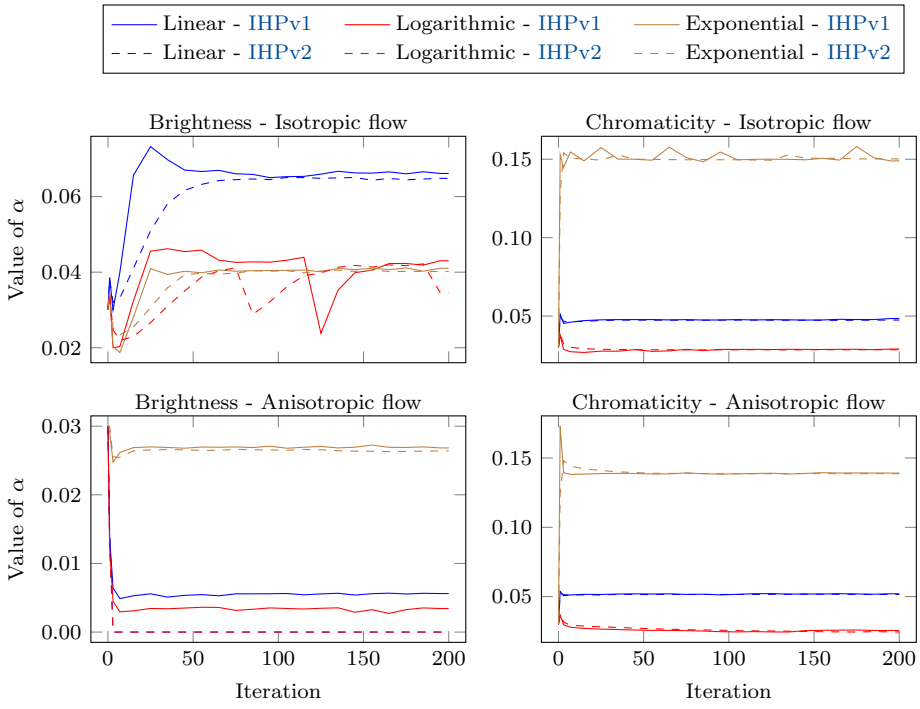


Figure 2.9: Evolution of α for different kind of workloads and versions of IHP on Desktop system. Single-precision computations.

As shown in Figure 2.11, in the Laptop system, the CPU can handle much more computations. It is also more difficult to witness the differences in behaviour between IHPv1 and IHPv2 due to the smaller number of time steps. Note that in the first iterations both versions behave similarly since there is not enough data for IHPv2

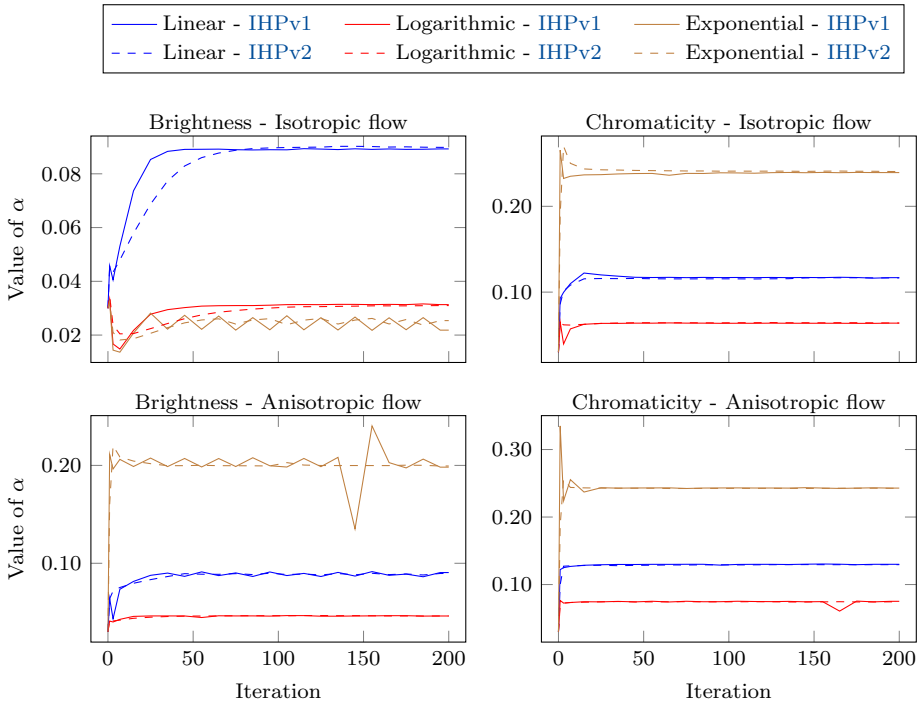


Figure 2.10: Evolution of α for different kind of workloads and versions of IHP on Desktop system. Double-precision computations.

to build a meaningful performance model of the processors. Despite that, it can be observed that the changes for α are steeper in IHPv1 than IHPv2.

Given the importance of energy consumption in devices, it is important to evaluate the impact of the proposed techniques in this regard. Tables 2.1 to 2.2 show the result regarding energy consumption.

In most scenarios, using only the GPU is the most efficient option in terms of energy. Despite that the heterogeneous implementations run faster, they are ballasted by the highly demanding CPUs. Take, for example, the linear workload using isotropic flow and double-precision arithmetic. The CPU handles approximately 10 % of the workload but increases power demands by 30 %.

It is worth noting that heterogeneous implementations might improve energy consumption in exponential workloads, where the large improvement in execution times compensates for the high consumption of the CPU cores.

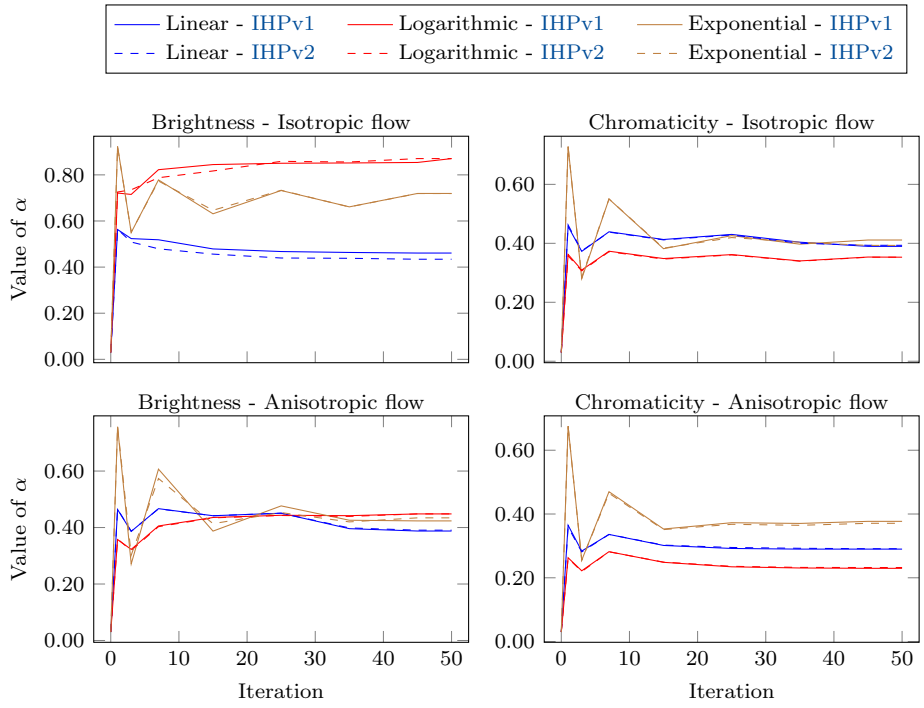


Figure 2.11: Evolution of α for different kind of workloads and versions of IHP on Laptop system. Single-precision computations.

		CPU-only	GPU-only	IHPv1	IHPv2
Isotropic flow	Linear	4.04	**0.48**	0.67	0.66
	Logarithmic	2.04	**0.27**	0.40	0.40
	Exponential	19.39	**2.11**	2.35	2.37
Anisotropic flow	Linear	17.32	**1.19**	1.76	1.63
	Logarithmic	8.75	**0.72**	0.94	0.90
	Exponential	79.11	**6.09**	7.05	6.62

Table 2.1: Energy consumption (Wh) for different implementations and different workloads on Desktop system. Single-precision computations. Lower is better.

		CPU-only	GPU-only	IHPv1	IHPv2
Isotropic flow	Linear	3.62	**0.95**	1.28	1.24
	Logarithmic	1.86	**0.52**	0.70	0.71
	Exponential	21.31	4.35	4.15	**3.78**
Anisotropic flow	Linear	16.42	**3.77**	4.78	4.86
	Logarithmic	6.89	**1.95**	3.10	2.67
	Exponential	81.21	18.55	**16.80**	16.83

Table 2.2: Energy consumption (Wh) for different implementations and different workloads on Desktop system. Double-precision computations. Lower is better.

2.5.1 Locally, everything is linear

Despite that considering the whole picture the workloads shown in Figure 2.6 are different, locally they can be considered linear. Figure 2.12 shows the CPU execution time per iteration after recalculating the CPU workload share, α, ten and twenty times. In this figure, the workload models and the share are calculated with the last $n = 10$ pieces of data. Note that the first 10 values of α are slightly scattered, since the algorithm is still looking for the optimal workload share, but it quickly converges. Once it converges, it is difficult—to say the least—to discern whether the workload fits a linear, logarithmic or exponential model.

2.6 Comparison of IHP against other libraries

As detailed in [67], IHP has been compared against LogFit [60, 61] and Concord [59], all using Intel TBB and OpenCL. This comparison has been performed in the Desktop system, using the image-denoising algorithms as a use case.

For the sake of simplicity, IHPv1 has been used in the comparison, since the workload of the experiment is linear and, as shown in Section 2.5, there is no significant difference between IHPv1 and IHPv2 in this kind of scenarios.

Concerning diffusion methods parameters, 1000 iterations have been computed, for both brightness and chromaticity, with $c_{\min} = 5$ and $c_{\max} = 50$. Following (2.1), initially, $\alpha = 0.03$. Also, the overlap region in our experiments is two rows in the tests. This is the smallest overlap region size that allows using centred differences for the approximations in the domain borders.

Tables 2.3 and 2.4 show the results of the comparison of IHPv1—using OpenCL and Intel TBB—with LogFit and Concord.

The main differences between IHPv1 and the other libraries lay in two details. First, IHPv1 requires less data movement between CPU and GPU. While LogFit and Concord spend several seconds in the best cases, IHP just spends up to 0.62 s in copy

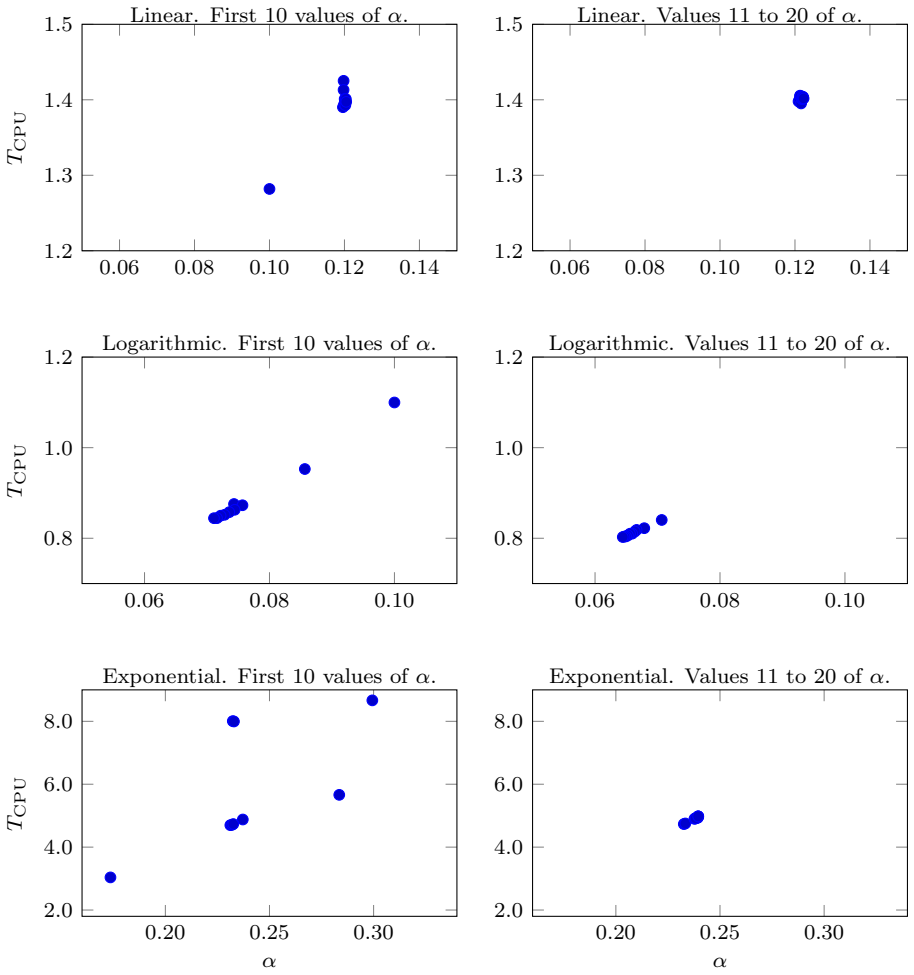


Figure 2.12: Representation of CPU execution times per iteration for different values of α.

		Brightness		Chromaticity	
		Isotropic	Anisotropic	Isotropic	Anisotropic
OpenCL		**14.00**	**18.42**	**40.70**	**81.56**
LogFit	CPU	9.01	25.81	54.62	112.85
	GPU	6.67	23.29	36.70	97.81
	Copy	12.02	21.60	59.74	61.49
	Total	**22.90**	**47.82**	**100.13**	**171.79**
Concord	CPU	12.98	127.48	70.42	154.78
	GPU	3.32	4.04	8.74	20.62
	Copy	7.37	3.73	15.61	14.68
	Total	**16.65**	**131.04**	**79.34**	**165.58**
IHPv1	CPU	5.78	17.12	32.73	68.44
	GPU	6.12	16.65	32.38	69.30
	Copy	0.18	0.23	0.30	0.26
	Total	**6.37**	**17.84**	**33.99**	**70.15**

Table 2.3: Execution times (in seconds) of OpenCL only, LogFit, Concord and IHPv1 implementations using single-precision. Lower is better.

		Brightness		Chromaticity	
		Isotropic	Anisotropic	Isotropic	Anisotropic
OpenCL		**16.39**	**30.78**	**57.37**	**125.54**
LogFit	CPU	21.19	49.68	113.09	182.83
	GPU	4.21	22.99	42.55	100.50
	Copy	18.10	43.21	96.26	128.69
	Total	**29.63**	**77.07**	**153.08**	**254.95**
Concord	CPU	28.58	156.51	272.46	1058.29
	GPU	8.35	24.31	41.92	83.55
	Copy	27.35	29.71	91.35	85.51
	Total	**41.67**	**172.33**	**291.51**	**1103.94**
IHPv1	CPU	9.16	27.65	44.76	107.59
	GPU	9.54	27.79	45.58	106.03
	Copy	0.48	0.52	0.62	0.62
	Total	**9.86**	**28.46**	**46.77**	**112.96**

Table 2.4: Execution times (in seconds) of OpenCL only, LogFit, Concord and IHPv1 implementations using double-precision. Lower is better.

operations. This behaviour was expected as LogFit and Concord are general-purpose libraries, while IHP is specifically designed to address iterative problems. Second, a much more accurate workload balance is found with IHPv1, so devices spend less time idle, 5 % in the worst case. For LogFit and Concord, non-optimal workload balances cause big differences in execution times, leaving GPU computational resources unused for long periods. This is particularly critical in Concord, which does not change the workload balance after the profiling phase, while LogFit continuously tries to find the optimal balance using a mechanism split into several phases.

Figure 2.13 shows the evolution of parameter α throughout the execution for IHPv1, LogFit, and Concord. Despite α being a parameter of IHP, it can be considered for LogFit and Concord too as the amount of work computed by CPU. These data have been obtained in fixed chunks of five iterations.

IHPv1 and LogFit methods converge to a value for the parameter α in a few iterations and keep it stable from very early on with single-precision operands. This is not the case with double-precision computations, where LogFit shows some instabilities. As mentioned before, Concord does not change the value of α after the initial profile phase, so its charts are completely flat.

As shown in Figure 2.14, IHPv1 manages to keep both devices busy most of the time. Note that differences between CPU and GPU execution times are around 10^{-2} seconds, being considerably higher only in the first iterations, where an optimal value of α is yet to be found.

Finally, Figure 2.15 shows the performance overhead induced by copy operations for different chunk sizes. Larger chunk sizes reduce the amount of time spent in data movement, which is a result to be expected. Even though, as discussed in Section 2.2, the size of the chunks should be carefully selected to find a tradeoff between the reduction of data movement and the rate at which IHP recalculates the workload balance.

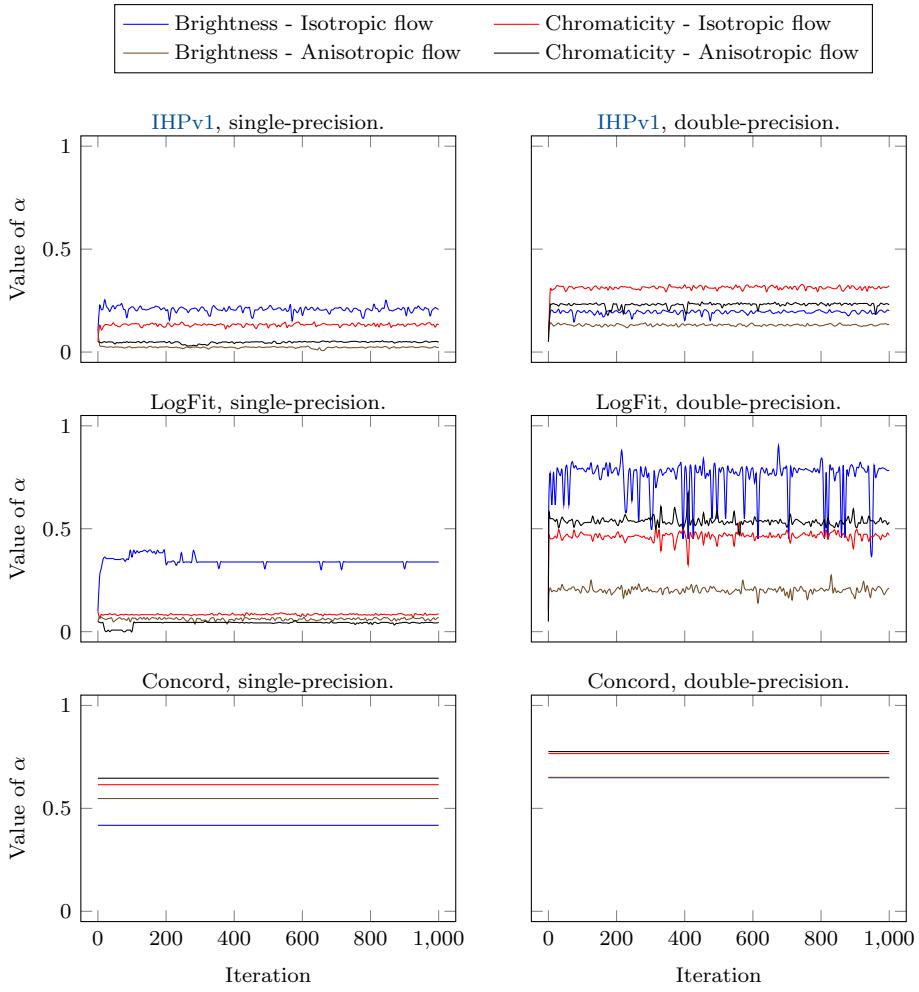


Figure 2.13: Evolution of α.

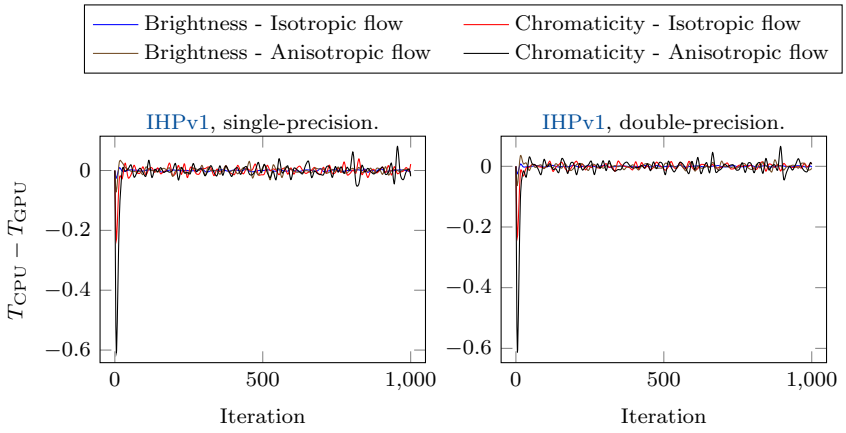


Figure 2.14: Time differences evolution.

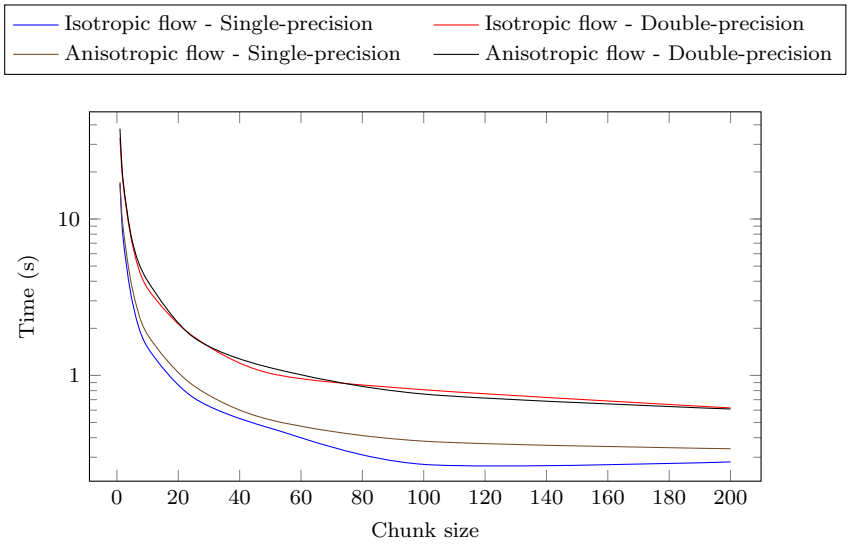


Figure 2.15: Chunk copy times.

The most significant contributions of this chapter are published and extracted from the subsequent articles:

- R. Laso, J. C. Cabaleiro, F. F. Rivera, M. C. Muñiz, and J. A. Álvarez-Dios, "Load balanced heterogeneous parallelism for finite difference problems on image denoising", *Computational and Mathematical Methods*, vol. 3, no. 3, e1089, 2021. DOI: https://doi.org/10.1002/cmm4.1089. *'Reproduced with permission from Springer Nature'.*
- R. Laso, J. C. Cabaleiro, F. F. Rivera, M. C. Muñiz, and J. A. Álvarez-Dios, "IHP: A dynamic heterogeneous parallel scheme for iterative or time-step methods—image denoising as case study", *The Journal of Supercomputing*, vol. 77, no. 1, pp. 95–110, Jan. 2021. DOI: https://doi.org/10.1007/s11227-020-03260-8.

Chapter 3

Thanos: a user space tool for thread and memory migration

With great power there must also come – great responsibility!
— Stan Lee, *The Amazing Fantasy #15.*

Focusing on the problem of efficient executions in NUMA systems, and considering the complexity of the problem described in Chapter 1, several algorithms have been developed to face the challenge of scheduling threads and memory pages in these complex architectures. This chapter explains how the current Linux scheduler works, gives a brief overview of the state of the art, and describes formally the algorithms developed during this thesis for both thread and memory migrations in NUMA systems.

3.1 Linux scheduler and scheduling in NUMA systems

Linux scheduler has been evolving through the years to improve its performance at a slow but continuous rate. Researchers and contributors to the kernel have proposed patches and modifications to address the existing issues of the scheduler, but also to address the new problems of incoming platforms.

3.1.1 Completely Fair Scheduler (CFS)

The current—at the time of writing this thesis—Linux scheduler algorithm is known as Completely Fair Scheduler (CFS) and was introduced in October of 2007. According to its developer, Ingo Molnar, "CFS basically models an *ideal, precise multi-tasking CPU* on real hardware" [74].

The objective of CFS is to perfectly balance processor time to each task or group of tasks. Ideally, with a processor that could be executing several tasks in parallel, and with 100 % computing power, each task would use an even amount of CPU power.

So, for two tasks running, each will be executed with the 50 % of the computational capabilities of the CPU. Since actual processors can only work with one task at a time, a *virtual runtime* measured in nanoseconds is used. This virtual runtime is the amount of time that the task would have used if it was executed in a perfect CPU.

CFS has a minimum time period defined, in which each thread should run at least once. This time is divided by the number of tasks to be scheduled to define the length of the time slices. Every time a process is preempted, ends its slice, the scheduler will take the process with the least virtual runtime to be executed during a time slice. It should be noted that the length of the time slices is pondered with the *niceness* of the processes. If the process has a high nice value, its slice will be shorter. If it has a low nice value, it will be granted more time. Furthermore, preempting processes incur overhead, so a minimum slice is defined to prevent switching contexts among processes too often [75].

3.1.2 Scheduling domains

Scheduling domains is a feature introduced in the scheduler of Linux to handle different layers of the system hierarchically.

Each scheduling domain spans one or several CPUs, which must be a superset of its child domains until each base domains span a single logical CPU [76]. Thanks to technologies like *hyperthreading*, a physical CPU can behave as several—typically, two or four—logical CPUs. Generally, the top of the hierarchy comprehends the full set of CPUs of the system. An example of a hierarchy is shown in Figure 3.1.

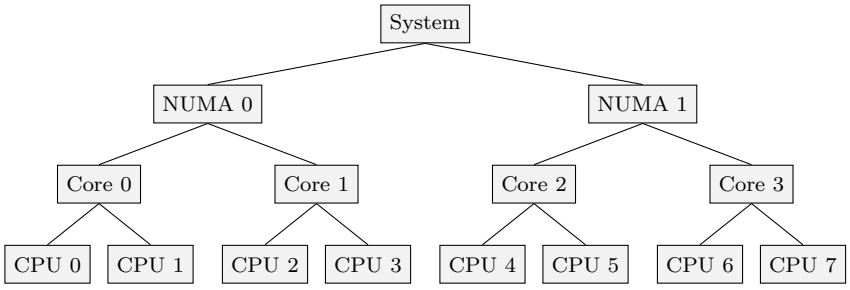


Figure 3.1: Example of scheduling domains for a system with two NUMA nodes, each node has one processor with two physical cores, and each core has two logical CPUs. Each node of the tree represents a scheduling domain that comprehends all its children. For example, NUMA node 0 comprehends CPUs 0, 1, 2, and 3.

The scheduler uses the domains to take care of the workload balance, so each level in the hierarchy is assigned a balancing policy which defines the periodicity at which the workload is balanced, among other parameters. So, periodically, the scheduler balances the workload on each domain based on a metric called *load*, which is the

combination of each thread's priority and its average CPU utilisation [77]. The load of each domain corresponds to the accumulation of the load of the threads running in that particular domain. If the load differs too much among domains in the same level of the hierarchy, some processes will be migrated from the busiest group to the less loaded one. It should be noted that the scheduler takes some factors into account, like cache affinity when migrating processes. A process whose cache is still valid is less likely to be migrated than another whose cache affinity is gone.

Imagine a NUMA system like the one shown in Figure 3.1. First, the scheduler will compute the workload of logical CPUs. Second, for each physical core, it will balance the workload for each logical CPU in the core. Third, for each NUMA node, it will balance the workload for each physical core. Finally, it will balance the workload among NUMA nodes.

Some notes about this procedure should be made [78]:

- Balancing among logical CPUs within the same core happens very often. There is no cache affinity since, typically, logical CPUs share the same last-level cache.
- Balancing among physical cores happens less frequently. Cache affinity should be taken into account, so processes are said to lose their affinity some milliseconds after being preempted.
- Balancing among NUMA nodes is even less frequent, and cache affinity lasts for longer since the cost of migrations among nodes is relatively high.

3.1.3 AutoNUMA and NUMA Balancing

In 2012, Andrea Archangeli proposed the AutoNUMA patch to the Linux scheduler, which aimed to improve both thread and memory placement in NUMA systems. AutoNUMA collects statistics about recent accesses to decide if a thread or a memory page is worth being migrated.

An array of counters is stored for each process to know which node was accessed the most in the recent past. When the time to be scheduled comes, that thread is moved to that node with more accesses.

Also, AutoNUMA introduced a mechanism for improving the locality of memory pages. With this mechanism, memory pages are periodically unmapped, causing a weak page fault the next time they are accessed. With the following non-local fault, the memory page will be queued to be migrated to that node [79]. To avoid moving pages too often, a quadratic filter is applied. That is, a page will be migrated only if it is accessed twice from the same NUMA node or by the same task [80].

Currently, this option is known as NUMA Balancing (NB) and is activated if the content of the file `/proc/sys/kernel/numa_balancing` is `1`.

3.1.4 Transparent Huge Pages (THPs)

Since 2003 the kernel of Linux uses a mechanism to group contiguous memory pages aiming to improve the performance of general systems, named *Huge Pages* [81]. The traditional meaning of huge—memory—pages is related to increasing the size of all memory pages in the system, improving performance in two ways. The first way is that the frequency of operations over the Translation Lookaside Buffer (TLB) is reduced, since each TLB miss comprehends a wider range of physical addresses, thus, it comprehends more data for the processes to work with. The second is that the number of entries in the TLB is greatly reduced so, typically, the TLB fits in the L2 cache, speeding up the operations over the TLB. Nevertheless, increasing the size of memory pages would increase the amount of reserved memory that is not finally used, somehow "wasting" some memory.

As shown in [82], the benefit of using Huge Pages varies with the workload. CPU intensive benchmarks might see their performance increased by between 7 % and 13 %. For other kinds of applications, like databases, improvements are lower, around 1 %.

In version 2.6.38, the kernel incorporated a patch by Andrea Arcangeli that introduced Transparent Huge Pages (THPs) [83]. The idea of this patch was to ease and extend the use of Huge Pages. Prior to this patch, a Huge Page had to be required and allocated explicitly in the code. After the patch, the system reserves some memory for using Huge Pages by either an explicit request or using an automated mechanism. The OS automatically tries to use a THP whenever a page fault occurs: if available, a THP is used and the contiguous small pages are grouped into the Huge Page; if not, the kernel falls back to default small pages as usual. Furthermore, a kernel daemon is periodically attempting to use free Huge Pages. If a Huge Page is free, the daemon scans allocated memory in small pages to be gathered. Finally, a Huge Page can be broken into small pages again under some operations like `mlock()` or `mprotect()`.

According to the benchmarks [84], using THPs can improve performance by up to 11 % with no effort from programmers or system administrators.

3.1.5 Issues of the current scheduler

By the time this thesis is being written, the scheduler presents one particular flaw regarding NUMA systems: it is focused on work balance rather than locality and affinity. The AutoNUMA patch tries to solve this problem and achieved great popularity. Nevertheless, manual mapping of memory and threads through `numactl` tools [85] are yet preferred to extract the most performance in NUMA systems. Though, it still requires the user to have a deep understanding of the behaviour of the applications and the characteristics of the system.

3.2 Thread and memory pages migration on kernel and user spaces

When designing a new mechanism or algorithm for thread and/or memory pages migration, there are two alternatives, implement it in kernel space or user space. Each approach has its advantages and disadvantages which should be carefully considered and discussed.

Algorithms implemented in kernel space benefit from being able to use all the information available in the system, like the CPU-time used, the current location of the memory pages, number and frequency of page faults, etc. Also, all threads and memory pages are under the control of the algorithm, with no restriction on permissions, so the strategy has full control over the system, and with great power, there must also come great responsibility. Another advantage is that it might act beforehand in resource allocation, deciding the initial placement of resources. This initial placement might be crucial in applications that require low response times. Nevertheless, working at a kernel level has some disadvantages too. Modifying the kernel of Linux might be a tough task due to its complexity, and expensive in terms of time to make the algorithms work, solve the possible bugs and so on. This time restriction is hard to follow for the researchers since it might slow down the research work. Also, the final users of the modules or patches might experience problems in terms of compatibility due to the kernel version or, mainly, permissions limitations.

On the other hand, user space tools have to deal with a limited amount of information to decide which is the optimal thread and memory mapping. Information might come from parsing the content of the `/proc` directory or the performance information from the performance monitoring unit (PMU). User space tools are also restricted to migrate user-level threads and memory pages due to limitations in permissions. Though, is not a big issue since kernel-level threads are often light in terms of computational demands. Also, user space algorithms are always a step behind the operating system in terms of resource allocations because the tools can reallocate resources only after the initial placement decided by the OS. This causes a twofold overhead: first, the required overhead of allocating the resources, and, second, the reallocation by the migration tool. On the advantages side, user space algorithms require less time to develop, since a lot of things can be handled back to the OS. This saves time in development and debugging, allowing the researchers to invest more time in improving the existing algorithms and developing new ones. User space tools are also more flexible in terms of compatibility since they can be independent of the kernel version used underneath. Finally, the biggest advantage of user space tools is that they usually do not require special permission to be executed, so the user can download the software, compile it and run it without asking for any further permissions which might not be granted.

3.3 Related work

A large number of articles have been written with proposals applying different solutions to solve the aforementioned flaws. These works can be categorised within kernel patches or modules and user space tools.

On one hand, most of the proposals aim for Linux kernel modifications or modules to improve memory or thread placement. Carrefour, by Dashti et al. [39], is a modification of the kernel to prevent and alleviate memory congestion for NUMA systems, improving performance up to 3.6× compared to the original kernel and other popular patches like AutoNUMA [86]. Carrefour utilises hardware counters to profile and measure performance, and it takes decisions to migrate memory pages, while the OS still decides the thread placement. In the work by Diener et al. [87] kernel Memory Affinity Framework (kMAF) is proposed, a kernel patch that improves thread and data affinity by analysing in runtime the shared and exclusive memory regions, migrating threads and memory pages. Achieving improvements in runtime of 13 % on average and up to 36 %. Multi-View Address Space (MVAS) [88], by Di Gennaro et al., is a kernel module that changes the mechanism that handles the page faults to improve the accuracy of per-thread memory working-set and migrate memory pages, increasing system performance by up to 40 %. Works by Chiang et al. [89–91] implement several modifications into the kernel to improve thread mapping, locality, and deal with memory congestion, achieving significant performance boosts in PARSEC 3.0 [38] benchmarks. Also, Gureya et al. [35] propose Bandwidth-Aware Page Placement (BWAP), an algorithm for memory pages placement based on asymmetric weighted page interleaving, combining an analytical performance model of the target system and online tuning, delegating thread mapping into the OS. With this approach, BWAP improves up to 66 % the performance of the system.

On the other hand, user space tools for thread and memory mapping is the least explored approach. Nevertheless, there are still some interesting works. AsymSched, by Lepers et al. [37], implements a dynamic thread and memory placement algorithm in Linux to improve performance, particularly, in asymmetric NUMA systems. These systems exhibit differences among interconnection buses, in terms of bandwidth, or some links might be unidirectional. Utilising information retrieved from hardware counters in runtime, AsymSched finds the best thread and memory location every second, focusing on maximising the bandwidth for communicating threads. This approach achieves significant performance improvements in single and multiple application workloads. Decongested locality (DeLoc) [92] is a tool that computes the optimal mapping after a profiling phase where the communication and memory data are gathered and recorded. Thus, the computed mapping aims to improve the data locality and reduce memory congestion. Authors of DeLoc claim that performance is improved by 61 % compared to the AutoNUMA Linux policy.

Table 3.1 shows a summary of the aforementioned works noting the advantages and disadvantages of each approach. The last column of the table shows the features of

Thanos—see Chapters 3 and 4—which is the main contribution of this work regarding scheduling in NUMA systems.

	Carrefour	kMAF	MVAS	Chiang	BWAP	AsymSched	DeLoc	Thanos
No kernel changes required	✘	✘	✓	✘	✓	✓	✓	✓
No kernel modules required	✓	✓	✘	✓	✘	✓	✓	✓
Runs in user/kernel space	Kernel	Kernel	Kernel	Kernel	Kernel	User	User	User
Uses Hardware Counters	✓	✘	✘	✘	✓	✓	✓	✓
No previous profiling required	✓	✓	✓	✓	✘	✓	✘	✓
Handles threads	✘	✓	✘	✓	✘	✓	✓	✓
Handles memory pages	✓	✓	✓	✘	✓	✓	✓	✓
Max. speedup (HIB)	3.60×	1.56×	1.66×	1.51×	1.66×	2.90×	1.61×	1.46×
Runtime overhead (LIB)	<5 %	<4 %	?	<2 %	<4 %	?	?	<8 %

Table 3.1: State of the art comparison. Acronyms HIB and LIB stand for "higher is better" and "lower is better", respectively.

3.4 Formulation

To explain in a formal way how the algorithms developed in this thesis work, and how they use the performance information gathered through hardware counters, some notation should be introduced in the first place.

A given NUMA system is characterised by N_{nodes} nodes. The k-th node, ν_k with $1 \leq k \leq N_{\text{nodes}}$, consists of Z_{k} cores named ζ_{kl}, $1 \leq l \leq Z_k$. At any time, there is set of p processes running, namely, $\Pi = \{\pi_1, \ldots, \pi_p\}$. Each process π_i comprehends a set of h_i threads named θ_{ij}, $1 \leq j \leq h_i$. The whole set of threads running at a given moment is noted as Θ. Furthermore, the processes of Π store their data across a set of memory pages, which will be called $\Psi = \{\psi_1, \ldots, \psi_m\}$.

Every T seconds, system information is gathered and processed to make decisions about the migrations to be done, so the intervals $\tau_1, \tau_2, \ldots$ can be defined.

In the decision-making process, several metrics can be considered, and they are represented by the following functions:

- $A(\theta_{ij}, \tau_t)$ represents the number of accesses to data in DRAM memory done by the thread θ_{ij} to data located in any node in the time interval τ_t.
- $A(\theta_{ij}, \nu_n, \tau_t)$ represents the number of memory accesses done by the thread θ_{ij} to data located in the node ν_n in τ_t.
- $\hat{A}(\psi_i, \nu_n, \tau_t)$ is the number of accesses performed by all the threads running in the node ν_n to data located in the memory page ψ_i in the interval τ_t.
- $L(\theta_{ij}, \nu_n, \tau_t)$ is the average latency of the memory operations performed by thread θ_{ij} while running in the node ν_n in τ_t.

- $\hat{L}(\nu_n, \nu_m, \tau_t)$ returns the average latency of the memory operations performed by threads running in the node ν_n to data stored in the node ν_m in τ_t.
- $\hat{L}(\psi_i, \nu_n, \tau_t)$ represents the average latency of the accesses performed by all threads running in the node ν_n to data stored in the memory page ψ_i in τ_t.
- $\hat{L}(\nu_n, \tau_t)$ is the average latency of the accesses performed by all threads running in the node ν_n to data located in any memory page in the time interval τ_t.
- $\hat{L}(\psi_i, \tau_t)$ is the average latency of the accesses performed by all threads in the system to data stored in the memory page ψ_i in τ_t.
- $\hat{L}(\tau_t)$ is the average latency of the accesses performed by all threads in the system to all memory pages in τ_t.
- $O(\theta_{ij}, \tau_t)$ is the number of operations performed by thread θ_{ij} in the interval τ_t.
- $O(\theta_{ij}, \nu_n, \tau_t)$ is the number of operations performed by thread θ_{ij} while running in the node ν_n in τ_t.
- $I(\theta_{ij}, \tau_t)$ is the operational intensity for thread θ_{ij} in the time interval τ_t computed like

$$I(\theta_{ij}, \tau_t) = \frac{O(\theta_{ij}, \tau_t)}{A(\theta_{ij}, \tau_t) \cdot S_{\text{cache}}}, \tag{3.1}$$

  where S_{cache} is the number of bytes of a cache line[1].
- $I(\theta_{ij}, \nu_n, \tau_t)$ is the operational intensity for thread θ_{ij} while running in the node ν_n. This operational intensity is computed as

$$I(\theta_{ij}, \nu_n, \tau_t) = \frac{O(\theta_{ij}, \nu_n, \tau_t)}{A(\theta_{ij}, \nu_n, \tau_t) \cdot S_{\text{cache}}}. \tag{3.2}$$

The set of thread migrations considered candidates to be performed in τ_t is defined as $M = \{M_1, M_2, \dots\}$. Each migration is denoted by the tuple $M_i = \left(\vec{\Theta}, \vec{Z}, Q\right)$, where $\vec{\Theta}$ is the list of threads to be migrated, $\vec{Z}$ their respective destination cores and Q is a given score of that migration.

Two types of migrations are considered, a single migration or an interchange, depending on whether one or two threads are involved. In the first case, with $\vec{\Theta} = [\theta_{ij}]$ and $\vec{Z} = [\zeta_{kl}]$, θ_{ij} would be simply moved to core ζ_{kl}. In an interchange, where $\vec{\Theta} = [\theta_{ij}, \theta_{i'j'}]$ and $\vec{Z} = [\zeta_{kl}, \zeta_{k'l'}]$, the thread θ_{ij} would be migrated to ζ_{kl} and thread $\theta_{i'j'}$ would be migrated to $\zeta_{k'l'}$.

[1] Assuming that for every access to data in DRAM, only S_{cache} bytes are copied into cache memory.

The migration of threads to NUMA nodes is possible as well. In that case, each migration is similarly defined like $M_i = \left(\vec{\Theta}, \vec{N}, Q\right)$, where $\vec{N}$ is the list of destination nodes for threads in $\vec{\Theta}$.

Finally, memory migrations are defined by the set $\hat{M} = \left\{\hat{M}_1, \hat{M}_2, \dots\right\}$. Each memory migration is represented by the tuple $\hat{M}_i = \left(\vec{\Psi}, \nu_n\right)$, where all the pages in $\vec{\Psi}$ would be migrated to node ν_n.

3.5 Thanos: the migration tool

As a result of this thesis, a novel tool for thread and memory migration has been developed, named Thread & memory migration Algorithms for NUMA Optimised Scheduling (Thanos) [93].

Thanos works in user space, migrating the threads and the memory pages of the target program—or script—aiming to improve its performance. Based on the information collected from different sources, primarily the hardware performance counters and the `/proc` folder, it takes decisions on-the-fly about which threads and pages should be migrated and where. Note that Thanos implements different algorithms for thread and memory migrations, which are described in Section 3.6. More algorithms can be implemented in the future, using the same common infrastructure.

Three stages are periodically executed in Thanos: measurement, processing, and decision-making for threads and memory pages.

3.5.1 Performance measurement

Several options can be considered for measuring and quantifying the performance of the system. These options vary from considering the current use of CPU-time, the number and frequency of page faults, etc. As mentioned in Section 1.1.1, an adequate way of quantifying the performance of a NUMA system is to use the metrics of the 3DyRM.

Nowadays processors include hardware performance counters (HC). These are special-purpose registers that store information on the activities of the processor such as retired instructions, cache misses, branch predictions, memory traffic and more. Compared to software profilers, HC provide a wider range of events and metrics with little overhead and without requiring any change on the target program. The number of hardware counters and the events that can be used is different for each specific model and each manufacturer.

In particular, the tool developed during this thesis uses the hardware counters of Intel processors[2], through the use of Intel Precise Event Based Sampling (PEBS) [94]

[2]The tool could be adapted to be used in other platforms such AMD o ARM processors.

and Perfmon [95]. Intel PEBS is a feature available in Intel processors that allow for the recording and collection of periodic samples containing the selected hardware counters. Perfmon gives an interface to simplify the extraction of the information from these hardware counters.

To compute the 3DyRM metrics, the following counters are monitored:

- `MEM_TRANS_RETIRED:LATENCY_ABOVE_THRESHOLD`: memory operations for which latency is above a given threshold. A threshold value is given as an option to the migration tool, which by default is 1, so every transaction can be sampled.
- `OFFCORE_REQUESTS:ALL_DATA_RD`: number of read requests off-core, that is, to data allocated in DRAM memory.
- `INST_RETIRED`: total number of instructions retired.
- `FP_ARITH:SCALAR_DOUBLE:SCALAR_SINGLE`: scalar single- and double-precision floating point operations executed.
- `FP_ARITH:128B_PACKED_DOUBLE`: 128–bit vector double-precision floating point operations executed.
- `FP_ARITH:128B_PACKED_SINGLE`: 128–bit vector single-precision floating point operations executed.
- `FP_ARITH:256B_PACKED_DOUBLE`: 256–bit vector double-precision floating point operations executed.
- `FP_ARITH:256B_PACKED_SINGLE`: 256–bit vector single-precision floating points operations executed.

For each core in the system and each hardware counter, a buffer is created. PEBS fills the buffer with the samples, and periodically, the samples are extracted and processed. The obtained information is used by the algorithms included in this chapter to compute and search for the optimal mapping.

3.5.2 Processing

Adopting the 3DyRM for optimising the performance allows to define three basic performance functions to be optimised:

$$P_{\mathrm{l}}(\theta_{ij}, \nu_n, \tau_t) := L(\theta_{ij}, \nu_n, \tau_t), \tag{3.3}$$
$$P_{\mathrm{o}}(\theta_{ij}, \nu_n, \tau_t) := O(\theta_{ij}, \nu_n, \tau_t), \tag{3.4}$$
$$P_{\mathrm{i}}(\theta_{ij}, \nu_n, \tau_t) := I(\theta_{ij}, \nu_n, \tau_t). \tag{3.5}$$

Equations (3.3), (3.4) and (3.5) focus only on improving average latency, the number of executed operations per second or operational intensity, individually.

Nevertheless, treating the problem as a single-objective optimisation problem [96, 97] is not enough due to the complexity of NUMA systems, so a combination of the three parameters should be used, resulting in a multi-objective optimisation problem. To combine these parameters, a scalarisation [98] can be introduced,

$$P\left(\theta_{ij}, \nu_n, \tau_t\right) = \frac{O\left(\theta_{ij}, \nu_n, \tau_t\right) \cdot I\left(\theta_{ij}, \nu_n, \tau_t\right)}{L\left(\theta_{ij}, \nu_n, \tau_t\right)}. \tag{3.6}$$

and the problem is turned back into a single-objective optimisation problem. Note that this function might be used as a fitness function in the optimisation problem of improving performance.

According to the temporal locality principle, recently accessed data is likely to be accessed again, so it is interesting to save the performance of a thread within a given node. Though, the chances of accessing the same data, supposed to be in the same node, are reduced rapidly with time. So a decay function should be applied:

$$f(t) = \exp\left(-t^p/d\right), \tag{3.7}$$

where t is the number of seconds since the performance measurement was taken, and p and d are parameters to tune the shape of the decay function. This is a particular case of the super-Gaussian function,

$$f(t) = A \exp\left(-\left(\frac{(t-t_0)^2}{2\sigma_t^2}\right)^P\right), \tag{3.8}$$

where $A = 1$, $t_0 = 0$, $p = 2P$ and $d = \left(2\sigma_t^2\right)^P$.

Figure 3.2 shows how the decay function behaves for several values of p and d. The selected values are $p = 3$ and $d = 30$, since it keeps the original value for a couple of seconds, but decreases fast past that time.

When processing the information of memory pages for their migration, an ageing factor is applied as well,

$$f(t_{\text{mig}}) = \frac{1}{1 + t_{\text{mig}}}, \tag{3.9}$$

where t_{mig} is the number of seconds until the next execution of a memory migration strategy. This ageing factor introduces the concept of temporal locality in the memory migration algorithms, giving more relevance to the most recent information.

Also, with the functions described in Section 3.4, the concept of the preferred node can be defined for both threads and memory pages, namely ν_{pref} and $\hat{\nu}_{\text{pref}}$. For a given thread θ_{ij}, its preferred node is the node in which θ_{ij} performs most of its memory operations. Formally,

$$\nu_{\text{pref}} = \arg\max_{\nu_r} A\left(\theta_{ij}, \nu_r, \tau_t\right). \tag{3.10}$$

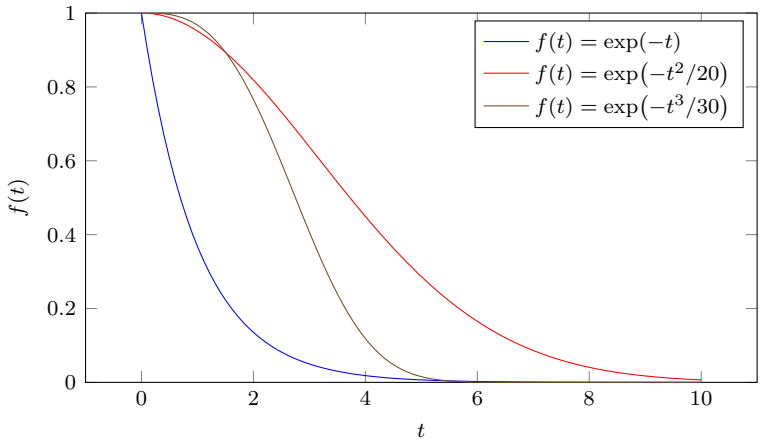


Figure 3.2: Examples of decay function, see (3.7), for several values of p and d.

Similarly, the preferred node of the memory page ψ_i is the node that hosts the threads that perform most memory accesses to data located in ψ_i. That is,

$$\hat{\nu}_{\text{pref}} = \arg\max_{\nu_r} \hat{A}\left(\psi_i, \nu_r, \tau_t\right). \tag{3.11}$$

Preferred nodes are a key element of performance in NUMA systems since they define the theoretical best node for a given thread or memory page.

It should be noted that the preferred node for a thread or a memory page might change during the execution. Think, for example, about a program solving a problem using a FEM. Several phases might take place, like reading the input data, building the system matrices, solving the required systems of equations, post-processing and writing the output. Each phase of the execution is likely to have a different memory access pattern, increasing the complexity of the problem of thread and memory mapping.

Fake Transparent Huge Pages (fTHPs)

Given the nature of Transparent Huge Pages—see Section 3.1.4—it is problematic for Thanos to work with them since in user space is not possible to know the THP which a small page belongs to. It is only possible to know whether a page belongs to a THP or not, given root permissions.

To overcome this limitation, the mechanism of THP is emulated in Thanos using fake Transparent Huge Pages (fTHPs). Memory regions of children processes are scanned through the information available in the `/proc` folder, so memory addresses

are grouped in fTHPs of a given size. By default, Thanos takes the size of a THP in the system, which is usually 512 small pages. Let be, for example, a memory region that comprehends the addresses between 0x100000 and 0x700000, and suppose that Thanos makes groups of 512 small pages of 4 kB. Thus, the region [0x100000, 0x700000) will be treated as 3 different fTHPs: [0x100000, 0x300000), [0x300000, 0x500000), [0x500000, 0x700000). Note that fake THPs can be smaller if required so, for example, if a memory region comprehends 200 pages, those pages are grouped in a fTHP of size 200 instead of size 512.

This mechanism also tries to solve two flaws of Thanos that will be further discussed in Chapter 4. First, the scarcity of information collected for each memory page is mitigated. Therefore, if the fTHPs has size 512, the number of samples per page received is expected to be multiplied by 512. Second, the impact of migrations is increased, since the migration of a fTHP implies the migration of a large group of small pages. Nevertheless, it is expected some loss in granularity since the small pages belonging to a fTHP might have different locality characteristics.

3.5.3 Decision making

Once the information has been collected and processed, the different algorithms implemented in Thanos can decide which threads and memory pages should be migrated and their respective destinations. These algorithms follow different approaches, which are described in detail in Section 3.6.

3.6 Migration algorithms

Ideally, the running threads and their data should be as close as possible at any moment. In a NUMA system, that would imply having the threads and the memory pages they access in the same NUMA node since that is the shortest possible path. As mentioned in Chapter 1, the accesses to data in remote nodes have a higher latency and a lower bandwidth than operations in the local node. But, as usual, reality shows that theory does not always work and an excess of local memory operations might cause other phenomena like memory contention and saturation of the interconnection buses [35]. In that scenario, the workload should be balanced across different NUMA nodes by migrating memory pages to other nodes, which might result in desirable remote accesses.

3.6.1 Thread migration algorithms

This section presents the collection of algorithms developed in this thesis for migrating threads. Each algorithm follows a different approach and has different characteristics. On one hand, some algorithms prioritise those threads with lower performance, taking the decisions following lottery-based rules, where the most promising migrations are

more likely to be selected. On the other hand, other algorithms look for a global improvement, taking into account the performance of all the threads, and potentially migrating a large number of threads at the same time.

CRA

The Completely Random Algorithm (CRA) serves as a validation for the rest of the algorithms.

Every T_{CRA} seconds, CRA is executed selecting a set of m random threads[3] named $\hat{\Theta}$. For each thread, $\theta_{ij} \in \hat{\Theta}$, a destination core, $\zeta_{k'l'}$, is selected also in a completely random way. In the case that $\zeta_{k'l'}$ was previously hosting another thread $\theta_{i'j'}$, an interchange is performed between θ_{ij} and $\theta_{i'j'}$.

Algorithm 3.1 shows the pseudo-code of CRA.

Algorithm 3.1 CRA migration strategy.

Input: Threads $\Theta = \{\theta_{ij}, i = 1, \dots, p, j = 1, \dots, h_i\}$.
Cores $Z = \{\zeta_{kl}, k = 1, \dots, N_{\text{nodes}}, l = 1, \dots, Z_k\}$.
Number of threads to be migrated, m.
Output: Migrations to perform $M = \{M_1, M_2, \dots, M_m\}$.

1: **procedure** CRA(Θ, Z, m)
2: $\hat{\Theta}$ = random set $\{\theta_{ij} \in \Theta\}$ ▷ Select m random threads.
3: $M = \varnothing$ ▷ Migrations to perform is an empty set at the beginning.
4: **for each** $\theta_{ij} \in \hat{\Theta}$ **do** ▷ Compute candidate migrations.
5: ζ_{kl} := core hosting θ_{ij}
6: $\zeta_{k'l'}$:= random core such that $\zeta_{k'l'} \neq \zeta_{kl}$
7: **if** $\zeta_{k'l'}$ was free during τ_t **then**
8: $M = M \cup ([\theta_{ij}], [\zeta_{k'l'}], 0)$ ▷ Migrate thread θ_{ij} to $\zeta_{k'l'}$.
9: **else**
10: $\theta_{i'j'}$ = random thread running in $\zeta_{k'l'}$ during τ_t
11: $M = M \cup ([\theta_{ij}, \theta_{i'j'}], [\zeta_{k'l'}, \zeta_{kl}], 0)$ ▷ Move θ_{ij} to $\zeta_{k'l'}$ and $\theta_{i'j'}$ to ζ_{kl}.
12: **return** M

LBMA

Lottery-Based Migration Algorithm (LBMA) [99] is a weighted lottery migration algorithm in which migrations are driven by a reduced set of heuristic rules.

Only two rules are considered for giving points:

[3] The parameter m is an input argument for Thanos.

- q_1 points are granted if destination core ζ_{kl} was free during τ_t. By default, $q_1 = 2$.
- q_2 points are assigned according to the NUMA distance of the destination node ν_k to the preferred node of θ_{ij} such that

$$q_2 = \hat{q}_2 \cdot \frac{d\left(\nu_k, \nu_k\right)}{d\left(\nu_k, \nu_{\text{pref}}\right)}, \tag{3.12}$$

  where $d\left(\nu_i, \nu_j\right)$ corresponds to the distance between nodes as returned by the system call `numa_distance(i, j)`, and $\hat{q}_2 = 4$ by default.

At the end of every time interval, LBMA selects a set of m random threads, such set is named $\hat{\Theta}$. For each thread $\theta_{ij} \in \hat{\Theta}$, which is running in the core ζ_{kl}, almost all possible destinations are considered. The cores in the node ν_k are not considered since they are expected to provide similar performance to ζ_{kl}.

For the rest of the cores, the score Q is computed using the aforementioned scoring system. In case that a core, ζ_{kl}, was hosting another thread $\theta_{i'j'}$ in τ_t, the score of migrating $\theta_{i'j'}$ is computed too. That way, the set of migrations in τ_t, $M = \{M_1, M_2, \dots\}$, is formed.

Once all possible migrations are computed, a weighted lottery selection process is performed. Each candidate migration is assigned a random number between 0 and Q, using a uniform distribution, such that migrations with higher scores will likely get a higher random number. Finally, those m migrations with the highest random value are performed.

Algorithm 3.2 shows the pseudocode of this migration strategy.

IMAR²

Interchange and Migration Algorithm with performance Record and Rollback (IMAR²) [99] refines the LBMA algorithm, also using weighted lottery-based algorithm.

The algorithm is executed every T seconds, being the value of T variable within the interval $[T_{\min}, T_{\max}]$ according to the global performance of the system as explained later.

Using the information provided by the 3DyRM, particularly the performance function P, the selection of threads in $\hat{\Theta}$ can be improved to select those with the worst relative performance instead of selecting them randomly.

Let $P_{\text{rel}}(\theta_{ij}, \tau_t)$ be the relative performance of a thread, defined such as

$$P_{\text{rel}}(\theta_{ij}, \tau_t) = \frac{P(\theta_{ij}, \tau_t)}{\bar{P}(\pi_i, \tau_t)}, \tag{3.13}$$

$$\bar{P}(\pi_i, \tau_t) = \frac{1}{h_i} \sum_{j=1}^{h_i} P(\theta_{ij}, \tau_t). \tag{3.14}$$

Algorithm 3.2 LBMA migration strategy.

Input: Set of threads $\Theta = \{\theta_{ij},\ i = 1, \ldots, p,\ j = 1, \ldots, h_i\}$.
Set of cores $Z = \{\zeta_{kl},\ k = 1, \ldots, N_{\text{nodes}},\ l = 1, \ldots, Z_k\}$.
Number of threads to be migrated m.
Output: Set of migrations to be performed $M = \{M_1, M_2, \ldots, M_m\}$.

1: **procedure** LBMA(Θ, Z, m)
2: $\hat{\Theta}$:= random set $\{\theta_{ij} \in \Theta\}$ ▷ Set of m threads randomly selected.
3: $\hat{M} = \varnothing$ ▷ Candidate migrations is an empty set at the beginning.
4: **for each** $\theta_{ij} \in \hat{\Theta}$ **do** ▷ Compute candidate migrations.
5: ζ_{kl} := core hosting θ_{ij}
6: ν_{pref} := preferred node of θ_{ij}
7: **for each** $\zeta_{k'l'} \in Z \mid k' \neq k$ **do** ▷ For each core in a different node, search for candidate migrations.
8: **if** $\zeta_{k'l'}$ is free **then**
9: $Q = q_1 + \hat{q}_2 \cdot \frac{d(\nu_{k'}, \nu_{k'})}{d(\nu_{k'}, \nu_{\text{pref}})}$
10: $\hat{M} = \hat{M} \cup ([\theta_{ij}], [\zeta_{k'l'}], Q)$ ▷ Single migration of θ_{ij} to $\zeta_{k'l'}$.
11: **else**
12: **for each** $\theta_{i'j'}$ running in ζ_k **do**
13: ν'_{pref} := preferred node of $\theta_{i'j'}$
14: $Q = \hat{q}_2 \cdot \frac{d(\nu_{k'}, \nu_{k'})}{d(\nu_{k'}, \nu_{\text{pref}})} + \hat{q}_2 \cdot \frac{d(\nu_k, \nu_k)}{d(\nu_k, \nu'_{\text{pref}})}$ ▷ Score of migrating θ_{ij} to $\zeta_{k'l'}$ and $\theta_{i'j'}$ to ζ_{kl}.
15: $\hat{M} = \hat{M} \cup ([\theta_{ij}, \theta_{i'j'}], [\zeta_{kl}, \zeta_{k'l'}], Q)$ ▷ Interchange, θ_{ij} to $\zeta_{k'l'}$ and $\theta_{i'j'}$ to ζ_{kl}.
16: M := Weighted Lottery Selection $\left(\hat{M}, m\right)$ ▷ Perform weighted selection process of candidates $\hat{M}$.
17: **return** M

This function allows fair comparisons between threads of different processes. Imagine two processes π_a and π_b, being π_a much more computationally intensive. Comparing by raw performance would always highlight processes of π_b for having low values.

Thus, $\hat{\Theta}$ will be formed by the n threads with the lowest relative performance—see equation (3.13)—and that their relative performance is under a given threshold δ_r, $0 < \delta_r < 1$. So, $\forall \hat{\theta}_{ij} \in \hat{\Theta}$, $P_{\text{rel}}(\hat{\theta}_{ij}, \zeta_{kl}, \tau_t) < \delta_r$.

A minimum threshold is necessary to prevent unnecessary migrations. Let us take a scenario in which the worst thread has a relative performance of 0.98. In that situation, it seems not appropriate to say that there is a thread with bad performance as all threads are performing almost equally so, probably, its mapping is already optimal or

close to it[4]. On the other hand, if the worst thread has a relative performance of 0.50 or less, it is likely that its mapping is not optimal and that a migration is convenient.

Candidate migrations are considered for each $\theta_{ij} \in \hat{\Theta}$ in a similar way as done in LBMA, and a score, $S(\theta_{ij}, \zeta_{kl}, \tau_t)$, is computed considering four heuristic rules:

- q_1 points are granted if a destination core $\zeta_{k'l'}$ was not hosting threads during τ_t. By default, $q_1 = 2$.
- q_2 points are assigned to the cores in the preferred node. By default, $q_2 = 4$.
- q_3 points are given according to the previous performance of θ_{ij} in the considered destination core $\zeta_{k'l'}$. Performance during τ_t is compared with the last performance measure obtained by θ_{ij} when running in a core in node k'. If the previous performance was better, $q_3 = 4$. If it was worse, $q_3 = 1$. Otherwise, $q_3 = 2$.
- q_4 points are given if a swap is considered between θ_{ij} and $\theta_{i'j'}$, and the relative performance of $\theta_{i'j'}$ is below δ_r, $P_{\text{rel}}(\theta_{i'j'}, \zeta_{k'l'}, \tau_t) < \delta_r$. By default, $q_4 = 3$.

Once there are candidate migrations for every $\hat{\theta}_{ij} \in \hat{\Theta}$, a weighted lottery process is used to select those m migrations to be performed. Algorithm 3.3 shows the pseudocode of the migration selection process of IMAR^2.

Finally, in τ_{t+1}, global performance is computed using equation (3.23) and it is compared to the performance in the previous interval. Three situations are considered:

- If $P_{\text{sys}}(\mu_{t+1}, \tau_{t+1}) > P_{\text{sys}}(\mu_t, \tau_t)$, performance has improved and migrations are considered successful, so a new time interval is selected as $T = \max(T/2, T_{\min})$, and migration process is accelerated—migrations will be performed more often.
- If $P_{\text{sys}}(\mu_{t+1}, \tau_{t+1}) < \delta_g P_{\text{sys}}(\mu_t, \tau_t)$, $\delta_g \in (0, 1)$, migrations implied a substantial loss of performance, so a rollback is performed—migrations are undone—and migration process is decelerated making $T = \min(2T, T_{\max})$.
- Otherwise, migrations are accepted and T keeps its previous value.

CIMAR

Core-aware Interchange and Migration Algorithm with performance Record (CIMAR) [100] is an evolution of IMAR^2, with the objective of improving its stability and consistency. There are three main differences between both algorithms: the frequency of migrations, the selection of migrations to be performed, and the removal of the rollback.

[4]It might happen—in the extreme case—that the mapping is the worst possible, but the chances of that happening without the intervention of the user are negligible.

Algorithm 3.3 IMAR^2 migration strategy.

Input: Set of processes $\Pi = \{\pi_1, \pi_2, \ldots, \pi\}$.
Set of threads $\Theta = \{\theta_{ij},\ i = 1, \ldots, p,\ j = 1, \ldots, h_i\}$.
Set of cores $Z = \{\zeta_{kl},\ k = 1, \ldots, N,\ m = 1, \ldots, C_k\}$.
Relative performance threshold δ_r.
Number of threads to be migrated m.
Output: Set of migrations to be performed $M = \{M_1, M_2, \ldots, M_m\}$.

1: **procedure** IMAR^2($\Pi, \Theta, Z, \delta_r, m$)
2: Adjust T according to the global performance of the system.
3: **if** conditions for a rollback are met **then**
4: Perform rollback of migrations performed in τ_{t-1}.
5: **return** $\varnothing$.
6: $\hat{\Theta} = \{\theta_{ij} \in \Theta \mid P_{\text{rel}}(\theta_{ij}, \zeta_{kl}, \tau_t) < \delta_r\}$ ▷ Select n threads with worst relative performance and $P_{\text{rel}} < \delta_r$.
7: $\hat{M} = \varnothing$ ▷ Candidate migrations is an empty set at the beginning.
8: **for** each $\theta_{ij} \in \hat{\Theta}$ **do** ▷ Compute candidate migrations.
9: ζ_{kl} := core hosting θ_{ij}
10: **for each** $\zeta_{k'l'} \in Z \mid k' \neq k$ **do** ▷ Search for candidate migrations in cores of different nodes.
11: **if** $\zeta_{k'l'}$ is free **then**
12: $Q = q_1 + S\left(\hat{\theta}_{ij}, \zeta_{k'l'}, \tau_t\right)$ ▷ Compute score for migrating θ_{ij} to $\zeta_{k'l'}$.
13: $\hat{M} = \hat{M} \cup \left([\theta_{ij}], [\zeta_{k'l'}], Q\right)$ ▷ Single migration of θ_{ij} to $\zeta_{k'l'}$.
14: **else**
15: **for each** $\theta_{i'j'}$ running in $\zeta_{k'l'}$ **do**
16: $Q = S\left(\theta_{ij}, \zeta_{k'l'}, \tau_t\right) + S\left(\theta_{i'j'}, \zeta_{kl}, \tau_t\right)$
17: $\hat{M} = \hat{M} \cup \left([\theta_{ij}, \theta_{i'j'}], [\zeta_{k'l'}, \zeta_{kl}], Q\right)$ ▷ Swap, θ_{ij} would moved to $\zeta_{k'l'}$, and $\theta_{i'j'}$ to ζ_{kl}.
18: M := Weighted Lottery Selection $\left(\hat{M}, m\right)$
19: **return** M

As in IMAR^2, those threads with the lowest relative performance are selected for migration. In the computation of the possible migrations for every $\theta_{ij} \in \hat{\Theta}$, the following condition is added in CIMAR: a migration of θ_{ij}, running in core ζ_{kl}, to the core $\zeta_{k'l'}$ can be considered if, and only if, it meets that

$$S(\theta_{ij}, \zeta_{k'l'}, \tau_t) > S(\theta_{ij}, \zeta_{kl}, \tau_t). \tag{3.15}$$

When considering an interchange between θ_{ij} and $\theta_{i'j'}$, running in cores ζ_{kl} and $\zeta_{k'l'}$, respectively, the condition is:

$$S(\theta_{ij}, \zeta_{k'l'}, \tau_t) + S(\theta_{i'j'}, \zeta_{kl}, \tau_t) > S(\theta_{ij}, \zeta_{kl}, \tau_t) + S(\theta_{i'j'}, \zeta_{k'l'}, \tau_t). \tag{3.16}$$

That is, only migrations with better scores than keeping threads still are considered. Furthermore, the lottery-based selection process is eliminated, so the m migrations with the highest scores are selected.

The aforementioned change has a deep impact on the behaviour of the algorithm. The exploration of new mappings is reduced since "non-promising" migrations are not even considered. For example, IMAR^2 might perform a migration with a low score that happens to be beneficial, but CIMAR would not. Thus, CIMAR is more conservative than IMAR^2. This is a matter of a trade-off between exploring new migrations and avoiding those which seem counterproductive.

Another important change of CIMAR over IMAR^2 is the elimination of the roll-back. This phase of IMAR^2 was introduced in the first place for undoing migrations which resulted in the worst performance. Since CIMAR tries to prevent migrations with expected low performance, the need for rollback is greatly reduced. Furthermore, when undoing migrations, the overhead of moving threads is paid twice: first for the original migration, and second for undoing it. This way, this potential penalty is avoided.

Finally, due to the more conservative nature of CIMAR over IMAR^2, and for sake of simplicity, the frequency of the migration process is constant so the algorithm is executed every T_{CIMAR} seconds.

The pseudocode of CIMAR is shown in Algorithm 3.4.

NIMAR

Node-aware Interchange and Migration Algorithm with performance Record (NIMAR) [100] is an algorithm built over CIMAR to improve and address its flaws. CIMAR might show problems regarding work balance, where two computationally intensive threads can share and stress a CPU even if there are other CPUs less loaded. To solve this, NIMAR does migrations to NUMA nodes instead of cores, entrusting the work balance within nodes to the operating system. Therefore, it is the OS that decides the particular core in which a thread will run. Work balance in OS is a well-studied field, so its reliability for this task is high since it has better information to do it correctly

Algorithm 3.4 CIMAR migration strategy.

Input: Processes $\Pi = \{\pi_1, \pi_2, \ldots, \pi_p\}$.
Threads $\Theta = \{\Theta_{ij}, i = 1, \ldots, p, j = 1, \ldots, h_i\}$.
Cores $Z = \{\zeta_{kl}, k = 1, \ldots, N_{\text{nodes}}, l = 1, \ldots, C_k\}$.
Relative performance threshold δ_{perf}.
Maximum number of migrations m.
Output: Migrations to perform $M = \{M_1, M_2, \ldots\}$.

1: **procedure** CIMAR($\Pi, \Theta, Z, \delta_{\text{perf}}, m$)
2: $\hat{\Theta} = \{\theta_{ij} \in \Theta \mid P_{\text{rel}}(\theta_{ij}, \nu_n, \tau_t) < \delta_{\text{perf}}\}$ ▷ Select m threads with worst rel. performance and such that $P_{\text{rel}} < \delta_{\text{perf}}$.
3: $\hat{M} = \varnothing$ ▷ Candidate migrations is an empty set at the beginning.
4: **for** each $\theta_{ij} \in \hat{\Theta}$ **do** ▷ Compute candidate migrations.
5: ζ_{kl} := core hosting θ_{ij}
6: $Q_{\text{ref}} = S(\theta_{ij}, \zeta_{kl}, \tau_t)$
7: **for** each $\zeta_{k'l'} \in Z \mid k' \neq k$ **do** ▷ For each core in a different node, search for candidate migrations.
8: **if** $\zeta_{k'l'}$ is free **then**
9: $Q = S(\theta_{ij}, \zeta_{k'l'}, \tau_t)$ ▷ Compute score for migration of θ_{ij} to $\zeta_{k'l'}$.
10: **if** $Q > Q_{\text{ref}}$ **then** ▷ If it is better to migrate than keeping θ_{ij} still...
11: $\hat{M} = \hat{M} \cup \{[\theta_{ij}], [\zeta_{k'l'}], Q\}$ ▷ Add a single migration of θ_{ij} to $\zeta_{k'l'}$ to the set of candidates.
12: **else**
13: **for** each $\theta_{i'j'}$ running in $\zeta_{k'l'}$ **do**
14: $Q = S(\theta_{ij}, \zeta_{k'l'}) + S(\theta_{i'j'}, \zeta_{kl}, \tau_t)$ ▷ Compute score for migrations of θ_{ij} and $\theta_{i'j'}$.
15: **if** $Q > Q_{\text{ref}} + S(\theta_{i'j'}, \zeta_{k'l'}, \tau_t)$ **then** ▷ If it is better to migrate than keeping θ_{ij} and $\theta_{i'j'}$ still...
16: $\hat{M} = \hat{M} \cup \{[\theta_{ij}, \theta_{i'j'}], [\zeta_{k'l'}, \zeta_{kl}], Q\}$ ▷ Swap, θ_{ij} would be moved to $\zeta_{k'l'}$, and $\theta_{i'j'}$ to ζ_{kl}.
17: $M :=$ m migrations in $\hat{M}$ with highest Q
18: **return** M

by working on kernel space. Thus, this algorithm presents a hybrid approach between user-space and kernel-space thread scheduling.

NIMAR algorithm is executed every T_{NIMAR} seconds and selects the set of threads to be migrated, $\hat{\Theta}$, in the same way as CIMAR and IMAR^2, selecting those threads with worse relative performance.

For each thread $\theta_{ij} \in \hat{\Theta}$, running in node ν_n, the rest of the nodes are considered for the destination, and a score is given to that migration. If the destination node $\nu_{n'}$ is hosting $Z_{n'}$ or more threads, an interchange is considered, and for every thread $\theta_{i'j'}$ running in $\nu_{n'}$, the score of the migration of $\theta_{i'j'}$ to ν_n is computed. Points are given in a similar way to CIMAR:

- q_1 points are granted if destination node ν_n was hosting less than Z_n threads during τ_t, it has free cores. By default, we set $q_1 = 2$.

- q_2 points are assigned according to the NUMA distance of the destination node $\nu_{n'}$ to the preferred node of θ_{ij} in the same way as shown in equation (3.17).

- q_3 points are given according to the previous performance of θ_{ij} in the considered destination node $\nu_{n'}$. Performance during τ_t is compared with the last performance measure obtained by θ_{ij} when running in a core in node $\nu_{n'}$. If the previous performance was better, q_3 is set to 4, whereas if it was worse, $q_3 = 1$. Otherwise, $q_3 = 2$.

- q_4 points are given if a swap is considered between θ_{ij} and a thread $\theta_{i'j'}$ currently running in the node $\nu_{n'}$, and $P_{\text{rel}}\left(\theta_{i'j'}, \nu_{n'}, \tau_t\right) < \delta_{\text{perf}}$. By default, we set $q_4 = 3$.

Also, migrations with a lower score than the one obtained by keeping the thread in its current location are discarded. Finally, the m migrations with the best score are performed.

Algorithm 3.5 shows the pseudocode of NIMAR.

SMA

Let $\mu_t : \Theta \rightarrow \{\nu_1, \ldots, \nu_{N_{\text{nodes}}}\}$ be the function that returns the node in which a thread is running in time interval τ_t and let $S(\theta_{ij}, \nu_n, \tau_t)$ be a function that given a thread, θ_{ij}, and a NUMA node, ν_n, returns a score such that the higher the score, the better performance is expected from θ_{ij} while running in ν_n.

The value of $S(\theta_{ij}, \nu_n, \tau_t)$ is assigned given the following heuristics:

- q_1 points are granted if destination node ν_n was hosting less than Z_n threads during τ_t, so it has free cores. By default, $q_1 = 2$.

Algorithm 3.5 NIMAR migration strategy.

Input: Processes $\Pi = \{\pi_1, \pi_2, \ldots, \pi_p\}$.
Threads $\Theta = \{\theta_{ij}, i = 1, \ldots, p, j = 1, \ldots, h_i\}$.
Nodes $N = \{\nu_n, n = 1, \ldots, N_{\text{nodes}}\}$.
Relative performance threshold δ_{perf}.
Number of threads to be migrated m.
Output: Migrations to perform $M = \{M_1, M_2, \ldots, M_m\}$.

procedure NIMAR($\Pi, \Theta, N, \delta_{\text{perf}}, m$)
  $\hat{\Theta} = \left\{\theta_{ij} \in \Theta \mid P\left(\theta_{ij}, \nu_n, \tau_t\right)/\bar{P}\left(\pi_i, \tau_t\right) < \delta_{\text{perf}}\right\}$ ▷ Select m threads with worst relative performance and such that $\hat{P} < \delta_{\text{perf}}$.
  $\hat{M} = \varnothing$ ▷ Candidate migrations is an empty set at the beginning.
  for each $\theta_{ij} \in \hat{\Theta}$ **do** ▷ Compute candidate migrations.
    $\nu_{n'} :=$ node hosting θ_{ij}
    $Q_{\text{ref}} = S\left(\theta_{ij}, \nu_n, \tau_t\right)$
    for each $\nu_{n'} \in N \mid n' \neq n$ **do** ▷ For each different node, search for candidate migrations.
      if $\nu_{n'}$ has free cores **then**
        $Q = S\left(\theta_{ij}, \nu_{n'}, \tau_t\right)$ ▷ Compute score for migration of θ_{ij} to $\nu_{n'}$.
        if $Q > Q_{\text{ref}}$ **then** ▷ If it is better to migrate than keeping θ_{ij} still...
          $\hat{M} = \hat{M} \cup \left(\left[\theta_{ij}\right], \left[\nu_{n'}\right], Q\right)$ ▷ Add a single migration of θ_{ij} to $\nu_{n'}$ to the set of candidates.
      else
        for each $\theta_{i'j'}$ running in $\nu_{n'}$ **do**
          $Q = S\left(\theta_{ij}, \nu_{n'}, \tau_t\right) + S\left(\theta_{i'j'}, \nu_n, \tau_t\right)$ ▷ Compute score for migrations of θ_{ij} and $\theta_{i'j'}$.
          if $Q > Q_{\text{ref}} + S\left(\theta_{i'j'}, \nu_{n'}, \tau_t\right)$ **then** ▷ If it is better to migrate than keeping θ_{ij} and $\theta_{i'j'}$ still...
            $\hat{M} = \hat{M} \cup \left(\left[\theta_{ij}, \theta_{i'j'}\right], \left[\nu_{n'}, \nu_n\right], Q\right)$ ▷ Swap, θ_{ij} would be moved to $\nu_{n'}$, and $\theta_{i'j'}$ to ν_n.
  $M :=$ m migrations in $\hat{M}$ with highest Q
  return M

- q_2 points are assigned according to the NUMA distance of the destination node ν_n to the preferred node of θ_{ij} such that

$$q_2 = \hat{q}_2 \cdot \frac{d(\nu_n, \nu_n)}{d(\nu_n, \nu_{\text{pref}})}, \tag{3.17}$$

  where $d(\nu_i, \nu_j)$ corresponds to the distance between nodes as returned by the system call `numa_distance(i, j)`, and $\hat{q}_2 = 4$ by default.

- q_3 points are given according to the previous performance of θ_{ij} in the considered destination node ν_n. Performance during τ_t is compared with the last performance measurement obtained by θ_{ij} when running in node ν_n. If the previous performance was better, $q_3 = 4$, whereas if it was worse, $q_3 = 1$. Otherwise, $q_3 = 2$.

Scoring is accumulative, so a thread that meets several requirements would accumulate the points. For example, the score of a thread θ_{ij} to be migrated to ν_n, meeting the requirements for q_1 and q_3, would be $S(\theta_{ij}, \nu_n, \tau_t) = q_1 + q_3$.

Thus, the score of a system can be computed as the sum of the scores of all threads:

$$S_{\text{sys}}(\mu_t, \tau_t) = \sum_{\theta_{ij} \in \Theta} S(\theta_{ij}, \mu_t(\theta_{ij}), \tau_t). \tag{3.18}$$

Using this scoring system, Score Maximisation Algorithm (SMA) tries to maximise the total score in the server, which can be stated as the following optimisation problem:

Problem 3.1. *Find* $\hat{\mu}_t : \Theta \to \{\nu_1, \ldots, \nu_{N_{nodes}}\}$ *such that*

$$\hat{\mu}_t = \arg\max_{\hat{\mu}_t} S_{sys}(\hat{\mu}_t, \tau_t) = \arg\max_{\hat{\mu}_t} \sum_{\theta_{ij} \in \Theta} S(\theta_{ij}, \hat{\mu}_t(\theta_{ij}), \tau_t). \tag{3.19}$$

Attending to the principle of temporal locality, it is possible to assume that threads will have similar memory patterns in the time periods τ_t and τ_{t+1}. Therefore, given a mapping $\hat{\mu}_t$ solution of Problem 3.1 for τ_t, it can be stated that exists $\hat{\mu}_{t+1}$, solution of Problem 3.1 for τ_{t+1}, such that $\hat{\mu}_{t+1} \in B(\hat{\mu}_t, \delta)$, being $B(\hat{\mu}_t, \delta)$ the ball in our solution space centred on $\hat{\mu}_t$ with a certain radius $\delta > 0$.

That is, an optimal mapping in τ_t should be optimal or near-optimal in τ_{t+1} as well. Therefore, SMA computes the solution of Problem 3.1 for τ_t, and applies that solution to the mapping to be used in τ_{t+1}.

To solve the optimisation problem, a simulated annealing (SA) [101] algorithm is used. Starting with the mapping μ_t used in τ_t, the algorithm iteratively searches for solutions that optimises the fitness function (3.18), following these steps:

Step 0: Algorithm starts with $i = 0$ and the initial solution $\mu^i = \mu_t^0 = \mu_t$. Also, the parameters for initial temperature and cooling are defined such that, $T_0 = 0.3$ and $\alpha = 0.97$, respectively.

Step 1: On the i-th step, a candidate solution $\hat{\mu}^i$ in the neighbourhood of μ^i is generated. The new mapping is generated by selecting a random thread θ_{ij} mapped to ν_n and random destination $\nu_{n'}$. If $\nu_{n'}$ has free cores, θ_{ij} is mapped to $\nu_{n'}$. If not, another thread $\theta_{i'j'}$ mapped to $\nu_{n'}$ is selected, so θ_{ij} is mapped to $\nu_{n'}$ and $\theta_{i'j'}$ is mapped to ν_n.

Step 2: The fitness of the candidate solution is computed:

$$S_{\text{sys}}(\hat{\mu}^i, \tau_t) = \sum_{\theta_{ij} \in \Theta} S(\theta_{ij}, \hat{\mu}^i(\theta_{ij}), \tau_t). \tag{3.20}$$

Step 3: Generate a random number $U_i \in [0, 1]$ following a uniform distribution. If

$$U_i \leq \exp\left[\frac{-\left[S_{\text{sys}}(\mu^i, \tau_t) - S_{\text{sys}}(\hat{\mu}^i, \tau_t)\right]^+}{T_i}\right], \tag{3.21}$$

the candidate solution is accepted and $\mu^{i+1} = \hat{\mu}^i$. Otherwise, $\mu^{i+1} = \mu^i$. That is, the candidate solution is accepted if its fitness is better. If not, it is accepted with a given probability that decreases with temperature T_i and how much worse is the candidate.

Step 4: Let $i = i+1$ and $T_{i+1} = \alpha T_i$. If the maximum number of iterations is reached, finish the algorithm. Otherwise, go to Step 1.

The algorithm finishes once a maximum number of iterations is reached or when the algorithm was not able to improve the best solution after a given number of iterations. Since it is assumed that the mapping for τ_t is already close to optimal, it is possible to define the maximum number of operations to be low. That way, a low execution time is also ensured.

Summarising, SMA searches for the mapping $\hat{\mu}_t$ that solves Problem 3.1, and applies it to the time interval τ_{t+1} under the assumption that an optimal mapping for τ_t will be near the optimal for τ_{t+1}.

Once the optimal mapping is computed, the list of migrations that should be performed is built. Let μ be the mapping used in τ_t and $\hat{\mu}_t$ the optimal mapping that should have been used. For each thread θ_{ij}, it is compared whether its destination has changed or not. If $\mu_t(\theta_{ij}) = \hat{\mu}_t(\theta_{ij})$, the destination of θ_{ij} has not changed and no further action is required. Otherwise, $\mu_t(\theta_{ij}) \neq \hat{\mu}_t(\theta_{ij})$, a migration $M_i = (\theta_{ij}, \hat{\mu}_t(\theta_{ij}))$ is added to the set of migrations to be performed.

A final condition is imposed to perform the migrations to prevent undesirable overhead. The expected performance improvement—as a percentage—should be greater than the number of threads to be migrated. Formally,

$$100 \times \left(\frac{P_{\text{sys}}\left(\hat{\mu}_t, \tau_t\right)}{P_{\text{sys}}\left(\mu_t, \tau_t\right)} - 1\right) > \text{card}(M). \tag{3.22}$$

For example, a mapping that required migrating 10 threads to improve performance by 1 % would not be performed. On the other hand, a mapping that is expected to improve the performance by 10 % by migrating only 2 threads would be applied.

Algorithm 3.6 shows the pseudocode of Score Maximisation Algorithm.

Algorithm 3.6 SMA migration strategy.

Input: Threads $\Theta = \{\theta_{ij}, i = 1, \dots, p, j = 1, \dots, h_i\}$.
Mapping used in τ_t, μ_t.
Output: Migrations to perform $M = \{M_1, M_2, \dots\}$.

```
1:  procedure SMA(Θ, μ_t)
2:      M = ∅                                   ▷ Migrations to perform starts empty.
3:      μ̂_t := arg max_μ S_sys(μ, τ_t)          ▷ Find the mapping solution of Problem 3.1.
4:      for each θ_ij ∈ Θ do
5:          if μ_t(θ_ij) ≠ μ̂_t(θ_ij) then       ▷ If the mapping of θ_ij has changed...
6:              M = M ∪ ([θ_ij], [μ̂_t(θ_ij)])   ▷ Add migration of θ_ij to μ̂_t(θ_ij).
7:      if 100 × (S_sys(μ̂_t, τ_t)/S_sys(μ_t, τ_t) − 1) > card(M) then
8:          return M                            ▷ Migrations are expected to be worth.
9:      else
10:         return ∅                            ▷ Considered better to keep thing still.
```

DyRMMA

Let $\mu_t : \Theta \to \{\nu_1, \dots, \nu_{N_{\text{nodes}}}\}$ be the function that returns the node in which a thread is running in time interval τ_t and the function

$$P_{\text{sys}}(\mu_t, \tau_t) = \sum_{\theta_{ij} \in \Theta} P(\theta_{ij}, \mu_t(\theta_{ij}), \tau_t) \tag{3.23}$$

that returns the performance, as defined in equation (3.6), of the system. 3DyRM Migration Algorithm (DyRMMA) tries to maximise (3.23), thus, tries to solve the following optimisation problem:

Problem 3.2. *Find the thread mapping* $\hat{\mu} : \Theta \to \{\nu_1, \dots, \nu_{N_{nodes}}\}$ *such that*

$$\hat{\mu}_t = \arg\max_{\hat{\mu}_t} P_{sys}(\hat{\mu}_t, \tau_t) = \arg\max_{\hat{\mu}_t} \sum_{\theta_{ij} \in \Theta} P(\theta_{ij}, \hat{\mu}_t(\theta_{ij}), \tau_t). \tag{3.24}$$

Under the assumption that an optimal solution of Problem 3.2 for time interval τ_t would be optimal or near-optimal for τ_{t+1}, this algorithm computes the optimal mapping for τ_t and applies it to τ_{t+1}.

As was done for SMA, a simulated annealing algorithm is used to solve Problem 3.2:

Step 0: At first, $i = 0$, and the initial solution $\mu^i = \mu_t^0 = \mu_t$ is defined. Also, the parameters for initial temperature and cooling are defined such that, $T_0 = 0.3$ and $\alpha = 0.97$, respectively.

Step 1: On the i-th step, a candidate solution $\hat{\mu}^i$ from μ^i is generated in the same way as in the SMA algorithm.

Step 2: The fitness of the candidate solution is computed. To do that, an estimation of what would have happened in τ_t with mapping $\hat{\mu}^i$ should be calculated using the following equations:

$$\hat{l}(\theta_{ij}, \hat{\mu}^i, \tau_t) = \frac{1}{A(\theta_{ij}, \tau_t)} \sum_{n=1}^{N_{\text{nodes}}} A(\theta_{ij}, \nu_n, \tau_t) \cdot L(\hat{\mu}^i(\theta_{ij}), \nu_n, \tau_t), \tag{3.25}$$

$$\hat{o}(\theta_{ij}, \hat{\mu}^i, \tau_t) = O(\theta_{ij}, \tau_t) \cdot \frac{L(\theta_{ij}, \tau_t)}{\hat{l}(\theta_{ij}, \hat{\mu}^i, \tau_t)}, \tag{3.26}$$

$$P_{\text{sys}}(\hat{\mu}^i, \tau_t) = \sum_{\theta_{ij} \in \Theta} \frac{\hat{o}(\theta_{ij}, \hat{\mu}^i, \tau_t) \cdot I(\theta_{ij}, \tau_t)}{\hat{l}(\theta_{ij}, \hat{\mu}^i, \tau_t)}. \tag{3.27}$$

First, the estimated average latency, $\hat{l}$, of every thread mapped with $\hat{\mu}^i$ is computed. Second, the estimated number of operations, $\hat{o}$, is calculated assuming that it will improve proportionally to the improvement in latency. Finally, the estimated system performance is computed using the estimated number of operations, the estimated average latency and the intensity, which is assumed to remain constant.

Step 3: Generate a random number $U_i \in [0, 1]$ following a uniform distribution. If

$$U_i \leq \exp\left[\frac{-\left[P_{\text{sys}}(\mu^i, \tau_t) - P_{\text{sys}}(\hat{\mu}^i, \tau_t)\right]^+}{T_i}\right], \tag{3.28}$$

the candidate solution is accepted and $\mu^{i+1} = \hat{\mu}^i$. Otherwise, $\mu^{i+1} = \mu^i$. That is, the candidate solution is accepted if its fitness is better. If not, it is accepted with a given probability that depends on the temperature T_i and how much worse is the candidate.

Step 4: Let $i = i+1$ and $T_{i+1} = \alpha T_i$. If the maximum number of iterations is reached, finish the algorithm. Otherwise, go to Step 1.

In a similar way than SMA, the following condition is imposed to apply the mapping $\hat{\mu}_t$:

$$100 \times \left(\frac{S_{\text{sys}}(\hat{\mu}_t, \tau_t)}{S_{\text{sys}}(\mu_t, \tau_t)} - 1\right) > \text{card}(M). \tag{3.29}$$

That is, the migrations will only be performed if the expected performance improvement is bigger than the number of threads to be migrated.

Algorithm 3.7 shows the pseudocode of this strategy.

Algorithm 3.7 DyRMMA migration strategy.

Input: Threads $\Theta = \{\theta_{ij}, i = 1, \ldots, p, j = 1, \ldots, h_i\}$.
Mapping used in τ_t, μ_t.
Output: Migrations to perform $M = \{M_1, M_2, \ldots\}$.

1: **procedure** DyRMMA(Θ, μ_t)
2: $M = \varnothing$ ▷ Migrations to perform starts empty.
3: $\hat{\mu}_t := \arg\max_{\mu} P_{\text{sys}}(\mu, \tau_t)$ ▷ Find the mapping solution of Problem 3.2.
4: **for each** $\theta_{ij} \in \Theta$ **do**
5: **if** $\mu_t(\theta_{ij}) \neq \hat{\mu}_t(\theta_{ij})$ **then** ▷ If the mapping of θ_{ij} has changed...
6: $M = M \cup [\theta_{ij}, \hat{\mu}_t(\theta_{ij})]$ ▷ Add migration of θ_{ij} to $\hat{\mu}_t(\theta_{ij})$.
7: **if** $100 \times (P_{\text{sys}}(\hat{\mu}_t, \tau_t)/P_{\text{sys}}(\mu_t, \tau_t) - 1) > \text{card}(M)$ **then**
8: **return** M ▷ Migrations are expected to be worth.
9: **else**
10: **return** $\varnothing$ ▷ Considered better to keep thing still.

3.6.2 Memory migration algorithms

The complexity of migrating memory pages is higher than migrating threads. On one hand, less information is available due to two reasons: only one of the used hardware counters gives information about the memory pages themselves (`MEM_TRANS_RETIRED`) and there are—usually—a lot of memory pages present in the system, so there are fewer samples per memory page. On the other hand, memory pages are more expensive to migrate than threads, so each migration should be precise and the penalty for wrong migrations is higher. Nevertheless, some relevant information can be extracted and several algorithms have been implemented.

RMMA

Random Memory Migration Algorithm (RMMA) serves for validation purposes for the rest of algorithms, similarly to CRA, but working with memory pages. Periodically, RMMA selects a set of m memory pages, named $\hat{\Psi}$. For each page $\psi \in \hat{\Psi}$, a random destination node is selected, ν_n, and ψ_i is migrated to ν_n.

Algorithm 3.8 shows the pseudo-code of RMMA.

Algorithm 3.8 RMMA migration strategy.

Input: Memory pages $\Psi = \{\psi_1, \psi_2, \ldots, \psi_p\}$.
NUMA nodes $N = \{\nu_k, k = 1, \ldots, N_{\text{nodes}}\}$.
Number of pages to be migrated, m.

Output: Migrations to perform $\hat{M} = \left\{\hat{M}_1, \hat{M}_2, \ldots, \hat{M}_m\right\}$.

1: **procedure** RMMA(Ψ, N, m)
2: $\hat{\Psi}$ = random set $\{\psi_i \in \Psi\}$ ▷ Select m random pages.
3: $M = \varnothing$
4: **for each** $\psi_{ij} \in \hat{\Psi}$ **do** ▷ Compute candidate migrations.
5: $\nu_k :=$ node hosting ψ_i
6: $\nu_{k'} :=$ random node such that $\nu_{k'} \neq \nu_k$
7: $\hat{M} = \hat{M} \cup ([\psi_i], [\nu_k])$ ▷ Migrate page ψ_i to $\nu_{k'}$.
8: **return** M

TMMA

Threshold Memory Migration Algorithm (TMMA) counts the number of accesses and their ratio to migrate memory pages to their preferred node. For a given memory page $\psi_i \in \Psi$, let r_{pref} be the ratio of operations done by threads located in the preferred node of ψ_i, $\hat{\nu}_{\text{pref}}$, such that

$$r_{\text{pref}} = \frac{\hat{A}(\psi_i, \hat{\nu}_{\text{pref}}, \tau_t)}{\sum_{n=1}^{N} \hat{A}(\psi_i, \nu_n, \tau_t)}. \tag{3.30}$$

Also, let δ_{TMMA} be

$$\delta_{\text{TMMA}} = \min\left\{\frac{2}{3}, \frac{2}{N_{\text{nodes}}}\right\}. \tag{3.31}$$

For each memory page $\psi_i \in \Psi$, the algorithm checks whether r_{pref} is greater than δ_{TMMA}, or not. That is, the algorithm checks if the preferred node produces a number of operations much greater than other nodes. If the page is already in its preferred node, there is nothing else to do with it.

The threshold δ_{TMMA} is necessary to prevent migrations that are probably not worth it. Imagine a page ψ_i whose vector of ratios is $[0.26, 0.25, 0.25, 0.24]$. Migrating it to its preferred node is unlikely to produce significant performance improvements, even less considering the expensive overhead of migrating memory pages. Furthermore, the computation of the preferred node of ψ_i might have been altered by the inherent noise of sampling.

Similarly to how cache prefetching works, TMMA tries to anticipate and move contiguous memory pages to exploit spatial locality. For each page ψ_i to be migrated

to a given node ν_{dest}, up to S_{preload} next consecutive pages might be migrated to $\hat{\nu}_{\text{pref}}$. By default, $S_{\text{preload}} = 8$. Until TMMA finds a page ψ_{i+j} with a different preferred node, or j reaches S_{preload}, pages $\psi_i, \ldots, \psi_{i+j}$ will be migrated to the node $\hat{\nu}_{\text{pref}}$ including those pages for which no information is available. For example, if the page ψ_{i+3} has another preferred node, only pages ψ_i, ψ_{i+1} and ψ_{i+2} will be migrated to $\hat{\nu}_{\text{pref}}$.

Algorithm 3.9 shows the pseudocode of TMMA.

Algorithm 3.9 TMMA migration strategy.

Input: Memory pages $\Psi = \{\psi_1, \psi_2, \ldots, \psi_p\}$.
NUMA nodes $N = \{\nu_k, k = 1, \ldots, N_{\text{nodes}}\}$.
Number of pages to be migrated, m.

Output: Migrations to perform $\hat{M} = \left\{\hat{M}_1, \hat{M}_2, \ldots, \hat{M}_m\right\}$.

1: **procedure** TMMA(Ψ, N, m)
2: $\quad\hat{M} = \varnothing$
3: $\quad$**for each** $\psi_i \in \Psi \mid \psi_i \notin \hat{M}$ **do**
4: $\quad\quad\hat{\nu}_{\text{pref}} := \arg\max_{\nu_r} \hat{A}(\psi_i, \nu_r, \tau_t)$
5: $\quad\quad\nu_k :=$ node hosting ψ_i
6: $\quad\quad r_{\text{pref}} := \hat{A}(\psi_i, \hat{\nu}_{\text{pref}}, \tau_t) / \sum_{n=1}^{N} \hat{A}(\psi_i, \nu_n, \tau_t)$
7: $\quad\quad$**if** $\nu_k \neq \hat{\nu}_{\text{pref}}$ **and** $r_{\text{pref}} > \delta_{\text{TMMA}}$ **then**
8: $\quad\quad\quad\vec{\Psi} := \{\psi_i\}$
9: $\quad\quad\quad$**for** $j = 1, \ldots, S_{\text{preload}}$ **do** ▷ Compute preload.
10: $\quad\quad\quad\quad\nu'_{\text{pref}} := \arg\max_{\nu_r} \hat{A}(\psi_{i+j}, \nu_r, \tau_t)$
11: $\quad\quad\quad\quad$**if** $\nu'_{\text{pref}} = \hat{\nu}_{\text{pref}}$ or $\nu'_{\text{pref}} = \varnothing$ **then** ▷ If preload is feasible...
12: $\quad\quad\quad\quad\quad\vec{\Psi} := \vec{\Psi} \cup \psi_{i+j}$ ▷ Add ψ_j to the set of pages to migrate.
13: $\quad\quad\quad\quad$**else** ▷ No more pages available for preload.
14: $\quad\quad\quad\quad\quad\hat{M} = \hat{M} \cup \left[\vec{\Psi}, \hat{\nu}_{\text{pref}}\right]$ ▷ Pages in $\vec{\Psi}$ will be migrated to $\hat{\nu}_{\text{pref}}$.
15: $\quad\quad\quad\quad\quad$End preload and continue on line 3.
16: $\quad$**return** $\hat{M}$

LMMA

Latency Memory Migration Algorithm (LMMA) [100] searches for those memory pages that present latency issues and move them to their preferred node, or the least-busy node.

First, the algorithm decides which nodes are busy. It is said that the NUMA node

ν_n is busy whenever

$$\frac{\hat{L}\left(\nu_n, \tau_t\right)}{\hat{L}\left(\tau_t\right)} > \delta_{\text{busy}}. \tag{3.32}$$

By default, $\delta_{\text{busy}} = 1.3$. Also, the least busy node, ν_{alt}, is computed:

$$\nu_{\text{alt}} = \arg\min_{\nu_n} \frac{\hat{L}\left(\nu_n, \tau_t\right)}{\hat{L}\left(\tau_t\right)}. \tag{3.33}$$

For each page $\psi_i \in \Psi$, its average latency is compared to the global average latency. If

$$\frac{\hat{L}\left(\psi_i, \tau_t\right)}{\hat{L}\left(\tau_t\right)} > \delta_{\text{lat}}, \tag{3.34}$$

the page will be considered for migration since its latency is considerably higher than the average. By default, $\delta_{\text{lat}} = 1.3$. Given the case, two possible destinations are considered for ψ_i, its preferred node or the least saturated node. If the preferred node, $\hat{\nu}_{\text{pref}}$, is not busy, then $\nu_{\text{dest}} = \hat{\nu}_{\text{pref}}$, else, $\nu_{\text{dest}} = \nu_{\text{alt}}$. The least saturated node will only be its destination when the preferred node is noted as busy.

LMMA also performs preload, in the same way as TMMA (see Section 3.6.2).

Algorithm 3.10 shows the pseudocode of LMMA.

Algorithm 3.10 LMMA migration strategy.

Input: Memory pages $\Psi = \{\psi_1, \psi_2, \ldots, \psi_p\}$.
NUMA nodes $N = \{\nu_k, k = 1, \ldots, N_{\text{nodes}}\}$.
Number of pages to be migrated, m.

Output: Migrations to perform $\hat{M} = \left\{\hat{M}_1, \hat{M}_2, \ldots\right\}$.

1: **procedure** LMMA(Ψ, N, m)
2: $N_{\text{busy}} := \left\{\nu_n \in N \mid \hat{L}(\nu_n, \tau_t) / \hat{L}(\tau_t) > \delta_{\text{busy}}\right\}$ ▷ Evaluate busy nodes.
3: $\nu_{\text{alt}} := \arg\min_{\nu_r} \hat{L}(\nu_r, \tau_t) / \hat{L}(\tau_t)$ ▷ Pick the least busy node.
4: $\hat{M} = \varnothing$
5: **for each** $\psi_i \in \Psi \mid \psi_i \notin \hat{M}$ **do** ▷ Compute migrations.
6: **if** $\hat{L}(\psi_i, \tau_t) / \hat{L}(\tau_t) > \delta_{\text{lat}}$ **then** ▷ Move ψ_i if the latency is too high.
7: $\hat{\nu}_{\text{pref}} := \arg\max_{\nu_r} \hat{A}(\psi_i, \nu_r, \tau_t)$
8: **if** $\hat{\nu}_{\text{pref}} \in N_{\text{busy}}$ **then** ▷ Check if the preferred node is busy.
9: $\nu_{\text{dest}} := \nu_{\text{alt}}$
10: **else**
11: $\nu_{\text{dest}} := \hat{\nu}_{\text{pref}}$
12: $\vec{\Psi} := [\psi_i]$
13: **for** $j = 1, \ldots, S_{\text{preload}}$ **do** ▷ Compute preload.
14: $\hat{\nu}'_{\text{pref}} := \arg\max_{\nu_r} \hat{A}(\psi_{i+j}, \nu_r, \tau_t)$
15: **if** $\hat{\nu}'_{\text{pref}} = \nu_{\text{dest}}$ or $\hat{\nu}'_{\text{pref}} = \varnothing$ **then** ▷ If preload is feasible...
16: $\vec{\Psi} := \vec{\Psi} + \psi_{i+j}$ ▷ Add ψ_j to the set of pages to migrate.
17: **else** ▷ Else, no more pages available for preload.
18: $\hat{M} = \hat{M} \cup \left[\vec{\Psi}, \nu_{\text{dest}}\right]$ ▷ Pages in $\vec{\Psi}$ will be migrated to ν_{dest}.
19: End preload and continue on line 5.
20: **return** $\hat{M}$

The most significant contributions of this chapter are published and extracted from the subsequent articles:

- O. García Lorenzo, R. Laso Rodríguez, T. Fernández Pena, J. C. Cabaleiro Domínguez, F. Fernández Rivera, and J. Á. Lorenzo del Castillo, "A new hardware counters based thread migration strategy for NUMA systems", in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, E. Deelman, J. Dongarra, and K. Karczewski, Eds., Cham: Springer International Publishing, 2020, pp. 205–216, ISBN: 978-3-030-43222-5. DOI: https://doi.org/10.1007/978-3-030-43222-5_18.
- R. Laso, O. G. Lorenzo, F. F. Rivera, J. C. Cabaleiro, T. F. Pena, and J. A. Lorenzo, "LBMA and IMAR^2: Weighted lottery based migration strategies for NUMA multiprocessing servers", *Concurrency and Computation: Practice and Experience*, vol. 33, no. 11, e5950, 2021. DOI: https://doi.org/10.1002/cpe.5950.
- R. Laso, O. G. Lorenzo, J. C. Cabaleiro, T. F. Pena, J. Á. Lorenzo, and F. F. Rivera, "CIMAR, NIMAR, and LMMA: Novel algorithms for thread and memory migrations in user space on NUMA systems using hardware counters", *Future Generation Computer Systems*, vol. 129, pp. 18–32, 2022, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2021.11.008.

Chapter 4

Results on NUMA scheduling

I am inevitable.
— Thanos, *Avengers: Endgame.*

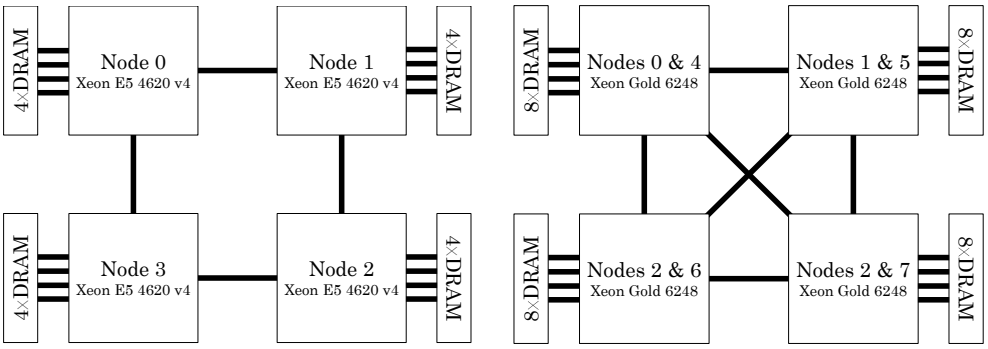
This chapter gathers an explanation of the experimental methodology, the environment, and the results obtained with the Linux scheduler and the proposals explained in Chapter 3. These experiments comprehend three different scenarios that represent the various ways of using NUMA systems. The results of Thanos are compared with CFS, which is used as a baseline, but also with other typical ways of mapping threads and memory pages. Furthermore, energy consumption is also discussed given its increasing importance in data centres.

4.1 Experimental environment

The experiments described in this chapter have been carried out in several NUMA servers with different topologies—see Figure 4.1—and features regarding memory latencies and bandwidth. Data corresponding latency and bandwidth matrices shown in Tables 4.1 and 4.2 were extracted with Intel Memory Latency Checker [103]:

- Server ctnuma1: a Linux system, kernel version 5.11.0 composed of four nodes with Intel Xeon E5-4620 v4 processors with 10 cores each, Broadwell-EP architecture, 25 MB L3 cache, 2.10 GHz–2.60 GHz, and 256 GB of DRAM. Server topology is shown in Figure 4.1(a). In this server, remote accesses have a latency more than 3× higher compared to local operations—see Table 4.1(a)—while its bandwidth is reduced by 79 %—see Table 4.1(b).

  Note that memory operations requiring 2-hop communications—for example, between nodes 0 and 2—have a slightly higher latency than the other remote accesses.



(a) Server ctnuma1. (b) Server ctnuma2.

Figure 4.1: Topology of the servers used in experimental validation.

Node	0	1	2	3
0	87.7	254.2	271.2	255.4
1	254.9	86.0	253.2	271.5
2	271.4	252.7	86.0	254.7
3	255.1	271.9	254.3	85.7

(a) Latency matrix (ns).

Node	0	1	2	3
0	61,255	12,709	12,002	12,368
1	12,153	61,176	12,210	12,007
2	12,028	12,395	61,288	12,689
3	12,371	11,992	12,704	61,261

(b) Bandwidth matrix (MB/s).

Table 4.1: Latency and bandwidth matrices for Server ctnuma1.

- Server ctnuma2: a Linux system, kernel version 4.18.0 composed of eight nodes with Intel Xeon Gold 6248 [104] with 10 cores each, Cascade Lake architecture, 27.50 MB L3 cache, 2.50 GHz–3.90 GHz, and 1 TB of DRAM. Server topology is shown in Figure 4.1(b). Note that there are four physical NUMA nodes, but eight nodes appear as a consequence of hyperthreading being enabled. All memory channels are in use and all memories are interconnected with each other, so all remote accesses have similar latencies. As reported by Tables 4.2(a) and 4.2(b), remote accesses have about 1.85× higher latency, while the bandwidth is decreased by 61 % approximately.

4.2 Benchmarks description

For the experimental validation of Thanos, several experiments have been performed using the NASA Advanced Supercomputing Parallel Benchmarks (NPB) [6] version 3.4.1, parallelised with OpenMP. With these benchmarks, it is expected to cover both

Node	0	1	2	3	4	5	6	7
0	83.8	134.3	147.6	134.0	89.7	144.7	138.5	142.4
1	134.7	78.2	133.9	144.7	143.6	85.0	142.4	137.6
2	145.3	132.2	78.3	134.9	137.1	140.7	84.4	143.4
3	132.4	144.6	134.0	76.7	141.4	136.3	142.5	84.1
4	84.2	136.7	141.2	135.7	77.8	146.6	133.9	144.2
5	136.2	85.0	134.9	139.7	145.3	77.2	144.1	133.6
6	140.7	133.7	85.0	136.2	133.7	143.4	77.0	145.8
7	134.8	139.6	136.1	87.6	145.1	132.8	144.7	83.4

(a) Latency matrix (ns).

Node	0	1	2	3	4	5	6	7
0	44,368	17,299	17,289	17,306	44,571	17,282	17,297	17,269
1	17,294	44,487	17,299	17,294	17,268	44,554	17,292	17,299
2	17,297	17,302	44,467	17,302	17,301	17,256	44,566	17,288
3	17,306	17,291	17,288	44,470	17,284	17,303	17,267	44,517
4	44,489	17,295	17,290	17,281	44,443	17,282	17,301	17,258
5	17,294	44,585	17,296	17,292	17,259	44,472	17,283	17,300
6	17,297	17,296	44,546	17,297	17,306	17,257	44,474	17,281
7	17,301	17,292	17,281	44,579	17,282	17,300	17,266	44,420

(b) Bandwidth matrix (MB/s).

Table 4.2: Latency and bandwidth matrices for Server ctnuma2.

CPU-limited and memory-limited programs. This is the list of benchmarks included in the suite:

- Block Tri-diagonal (BT): solver based on a computational fluid dynamics (CFD) pseudo-application on 3 dimensions. The application uses an approximate factorisation that decouples the x, y and z dimensions, resulting in block tri-diagonal systems of 5×5 blocks [6].

- Conjugate Gradient (CG): solver based on a CFD pseudo-application. The benchmark computes an approximation of the smallest eigenvalue of a large, sparse and unstructured matrix [6].

- Arithmetic Data Cube (DC): computations focused on data movement across cores [105]. This benchmark aims to stress all levels of memory, but also produces a large number of Input/Output (I/O) operations.

- Embarrasingly Parallel (EP): kernel designed to provide an estimate of the upper achievable limits of floating-point performance by generating pairs of Gaussian random deviates [6].

- 3-D Fast Fourier Transform (FT): computations with all-to-all communication. This benchmark computes three one-dimensional Fast Fourier Transformations based on a 3D FFT-based spectral method [6].

- Integer Sort (IS): algorithm with random memory accesses. This benchmark was designed for evaluating both computation speed and communication performance [106].

- Lower-Upper Gauss-Seidel (LU): pseudo-application solving a finite-difference discretization of the Navier-Stokes equations. It uses a symmetric successive over-relaxation (SSOR) method to solve a 7×7 block-diagonal system by splitting a Lower-Upper factorisation [6].

- Multi-Grid (MG): computations on a sequence of meshes, long- and short-distance communication. Memory intensive benchmark that computes the solution of a 3-dimensional scalar Poisson equation over coarse and fine meshes [6].

- Scalar Penta-diagonal (SP): solver based on a 3-D CFD pseudo-application based on a Beam-Warning factorisation [6].

- Unstructured Adaptive (UA): computations of an unstructured adaptive mesh with dynamic and irregular memory accesses. The benchmark solves a stylised heat transfer problem in a cubic domain, which is discretised using an adaptively refined and unstructured mesh [107].

The NPB have a class associated—W, S, A, B, C, D, E, F—which denote the size of the benchmark in question. Unless otherwise mentioned, class C has been used and, for example, an LU of class C is noted as LU.C.

Figure 4.2 shows the roofline model information obtained with Intel Advisor [108] for all the benchmarks used in the experiments with their amount of LLC misses as reported by `perf` [109]. With this information, it is possible to classify the benchmarks according to their rate of LLC misses:

- Low (0 % to 20 %): BT, EP, CG, and IS.

- Medium (20 % to 40 %): FT, MG and UA.

- High (40 % to 100 %): DC, LU and SP.

It should be noted that, after this preliminary analysis, the DC benchmark has been discarded since it is a benchmark limited on I/O operations, which are out the focus of this work and introduce an undesired high variability on the execution times.

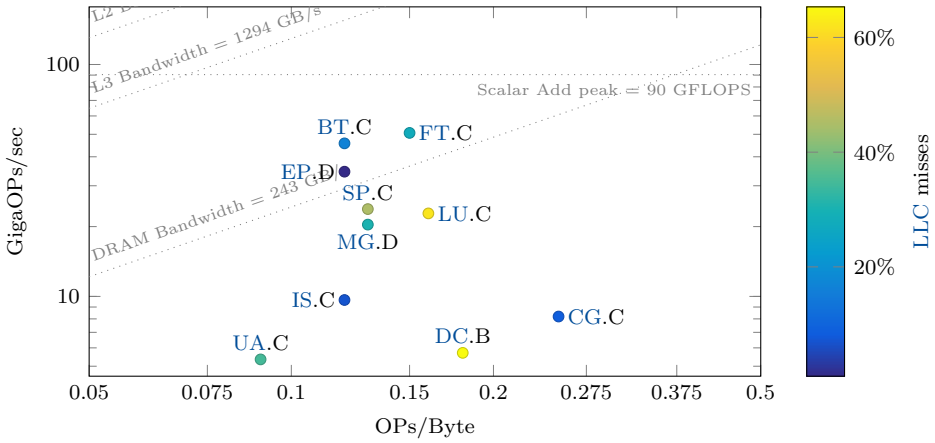


Figure 4.2: Roofline model of the NPB as reported by Intel Advisor in Server ctnuma1. The colour gradient shows the rate of LLC misses obtained with `perf`.

4.3 Experiments description

Experiments have been designed to match three common scenarios when using NUMA systems.

- Experiment Single: in this experiment, servers are used with one application at a time which can use all available resources, all cores and all available memory. Since the selected benchmarks are highly optimised, with high-quality code and well-programmed according to locality principles, little improvement could be expected, particularly in the EP and the IS which make very good use of the cache memory. Nevertheless, this experiment is interesting to give an idea of the potential of our migration tool in general and the different algorithms in particular.

- Experiment Interactive: servers are used interactively, that is, users can send little tasks at any time. This experiment emulates an interactive system based on anonymous data of the use of computing nodes in Centro de Supercomputación de Galicia (CESGA)[1]. The time at which each task starts to execute is fixed, so the objective is to reduce the execution time of each task. This experiment draws a scenario where the number of concurrent tasks changes with time, so migrations are expected to produce better results than in Experiment Single. The start time of each task is shown in Table 4.3. Note that start times are

[1] https://www.cesga.es

scaled down (×0.5) in Server ctnuma2 to achieve a similar pattern of concurrency compared to ctnuma1.

Task	ctnuma1	ctnuma2
LU.C-1	0.0	0.0
BT.C-1	18.0	9.0
CG.C-1	137.0	68.5
SP.C-1	190.0	95.0
LU.C-2	224.0	112.0
BT.C-2	244.0	122.0
CG.C-2	269.0	134.5
SP.C-2	316.0	168.0

Table 4.3: Start times in seconds for each task and server in Experiment Interactive.

Each task consists of 8 and 16 threads for Servers ctnuma1 and ctnuma2, respectively. At peak, it is possible to have more working threads than cores in the system. An example of an execution of this experiment is shown in Figure 4.3.

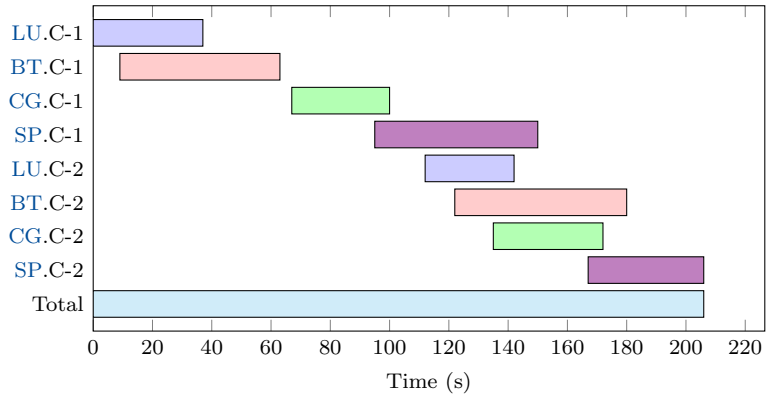


Figure 4.3: Example of a time trace for the Experiment Interactive.

Note that, in this experiment, the total execution time is mainly determined by the time at which the last benchmark is executed. This fades the possible improvements in the rest of the benchmarks and sets a lower bound in the test wall time.

- Experiment Queue: servers are used with a queue of tasks, like Slurm [45], where users send tasks and only a fraction of the resources are available. It emulates

as many users as NUMA nodes in the system, who send several tasks based on NPB. The objective is to reduce the time to complete all tasks, increasing the throughput of the system. There are granted only 10 cores per user for Servers ctnuma1 and ctnuma2. At peak, there are as many threads as cores, but all cores are in use until the end of the experiment. An example of an execution of this experiment is shown in Figure 4.4.

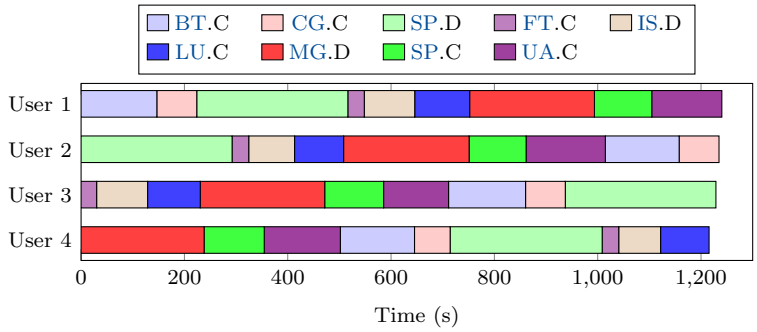


Figure 4.4: Example of a time trace for the Experiment Queue.

This is the experiment where the potential performance improvement due to thread and memory migrations is higher. Reducing the execution time of a task implies launching the following tasks earlier. Also, the chances of improving performance are greater due to the high number of processes simultaneously running in the system.

The results shown in this chapter are the average of five executions of each experiment in each server, with no warm-up phase.

4.3.1 Baseline comparison

To compare the different options available in Linux regarding scheduling in NUMA systems and define a baseline for the following sections, an analysis of Completely Fair Scheduler (CFS) with and without Transparent Huge Pages (THPs) and NUMA Balancing (NB) enabled has been done. To do so, experiments Single and Queue were performed in Server ctnuma1. Table 4.4 shows the execution times in the Experiment Single while normalised execution times—against using only CFS—are shown in Figure 4.5. The same data for Experiment Queue is shown in Table 4.5 and Figure 4.6, respectively.

When executing a single code in the system, the performance depends heavily on the benchmark and its kind of workload. The option that has a larger impact on

Algorithm	BT.C	CG.C	EP.D	FT.C	IS.D	LU.C	MG.D	SP.C	UA.C
CFS	42.6	23.4	73.3	8.75	30.9	34.2	161.3	46.7	50.8
CFS+NB	44.3	22.9	73.4	8.75	25.8	33.2	158.0	50.9	57.4
CFS+THP	37.6	17.2	73.6	10.61	45.3	33.2	145.8	27.2	32.2
CFS+NB+THP	41.6	20.8	74.0	8.77	26.4	48.9	78.9	37.0	47.3

Table 4.4: Execution times (s) for Experiment Single in Server ctnuma1. Lower is better.

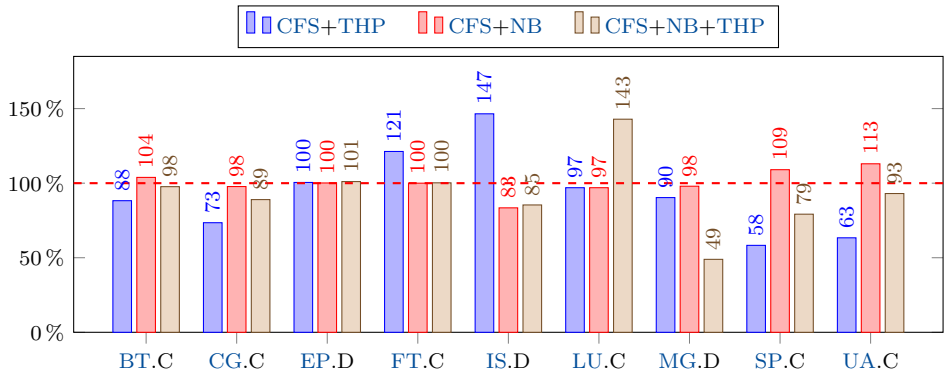


Figure 4.5: Normalised execution times (%) for Experiment Single in Server ctnuma1. Lower is better.

performance is THP. When enabling it, execution times might be drastically reduced—up to 42 % for SP.C—or increased—up to 47 % for IS.D. NUMA Balancing has lower influence, and its impact goes from −13 % to 17 %. It is when both options are enabled that the biggest differences can be seen. For example, the execution time of the MG.C benchmark is halved, but it is increased by 43 % for the LU.C.

Nevertheless, when executing several benchmarks simultaneously, the maximum performance is consistently obtained when enabling both THP and NUMA Balancing. Improvement varies from 5 % in the FT.C up to 47 % for particular benchmarks like CG.C or MG.D. This way, the total execution time is reduced by 27 % in the Experiment Queue.

Attending to the results, this chapter will refer to the Completely Fair Scheduler (CFS) with Transparent Huge Page (THP) and NUMA Balancing (NB) as Baseline from now onwards, unless otherwise stated.

For the sake of completeness, Thanos is compared not only with the Baseline but also with the Direct and the Interleave mappings via `numactl` [85], which are popular options for experienced users [110, 111]. With the Direct mapping, threads are pinned to a single NUMA node, and the memory is allocated in the same node.

Algorithm	BT.C	CG.C	EP.D	FT.C	IS.D	LU.C	MG.D	SP.C	UA.C	Total
CFS	165	130	259	99	113	120	349	210	155	1,756
CFS+NB	136	77	233	100	104	104	287	167	117	1,483
CFS+THP	163	134	258	99	114	119	336	193	150	1,692
CFS+NB+THP	137	69	235	94	79	94	192	132	123	1,274

Table 4.5: Execution times (s) for Experiment Queue in Server ctnuma1. Lower is better.

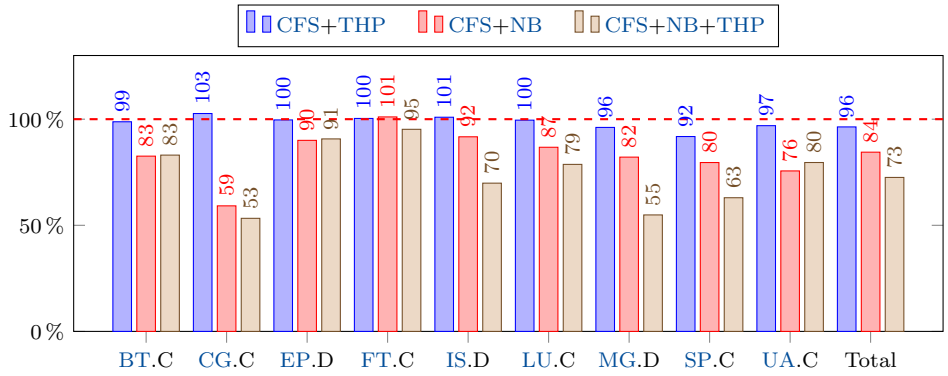


Figure 4.6: Normalised execution times (%) for Experiment Queue in Server ctnuma1. Lower is better.

In the Interleave mapping, memory pages are distributed across all NUMA nodes in a round-robin fashion. Note that using these options requires some knowledge of the system and the application, so they are only recommended for experienced users.

4.4 Precision of hardware performance counters

This section includes a brief study about the precision of the hardware performance counters (HC). Table 4.6 shows the giga-operations per second reported by the benchmarks and measured through HC for the NPB, as well as the relative error between both metrics.

Data show discrepancies of 4 % in the best scenario, though the error is typically in the range of 15 %–25 %, with a tendency of underestimating the real figure. Note that the biggest measurement errors happen when the reported operations per second are low, with an overestimation of several orders of magnitude.

Benchmark	Reported	Measured	Error (%)
BT.C	1.91	1.83	4.18
CG.C	0.22	0.37	67.98
EP.D	$4.71 \cdot 10^{-2}$	1.41	2,896.86
FT.C	1.10	0.95	14.13
IS.D	$2.13 \cdot 10^{-2}$	0.36	1,582.12
LU.C	1.72	1.26	26.95
MG.D	1.10	0.90	18.86
SP.C	1.11	0.89	19.49
UA.C	$4.80 \cdot 10^{-3}$	0.42	8,743.35

Table 4.6: Giga-operations per second reported by the benchmarks, measured through hardware performance counters, and relative error.

4.5 Experiment Single

This is the most traditional scenario and where the Baseline is expected to perform better. As mentioned before, low-performance improvements—if any—might be expected. Nevertheless, this experiment is still interesting to measure the potential of the algorithms proposed in this work.

4.5.1 Server ctnuma1

Table 4.7 shows the normalised execution times for the NPB. Raw execution times are shown in Table B.1 in Appendix B. First, it should be noted that Direct and Interleave mappings are suboptimal. In the case of Direct mapping, the memory is placed in a single memory node whenever possible, causing a lot of remote—and slow—memory operations. With the Interleave mapping, memory pages are scattered across the NUMA nodes without attending to any locality principle causing a large number of remote accesses.

Regarding the algorithms presented in this work, most of them manage to improve execution times, particularly those that handle thread migrations. Those that rely more on random processes cause losses of performance, like CRA and LBMA. For CRA, execution times increase up to 80 %.

IMAR^2 and DyRMMA achieve discrete results because of the same underlying problem. Both algorithms rely strongly on the metrics of the hardware counters in one or several phases. In the case of IMAR^2, it is due to the rollback phase which is triggered by calculating the global system performance based on equation (3.23), which is computed with the values extracted from hardware counters. If those values are not accurate, the rollback might be performed when it is not really required or *vice versa*. Figure 4.7 shows that CIMAR improves IMAR^2, mainly due to the removal of the

Algorithm	BT.C	CG.C	EP.D	FT.C	IS.D	LU.C	MG.D	SP.C	UA.C
Baseline	100	100	100	100	100	100	100	100	100
Direct	164	231	100	176	133	87	406	308	195
Interleave	114	103	101	111	107	58	216	150	119
CRA	111	99	100	101	109	61	184	143	123
LBMA	108	100	100	100	106	64	169	129	115
IMAR^2	100	91	99	98	103	58	128	115	103
CIMAR	93	77	100	99	97	59	78	90	99
NIMAR	92	80	100	99	98	58	79	90	93
SMA	92	81	100	99	98	62	78	89	99
DyRMMA	93	96	96	88	91	91	88	96	94
LBMA+TMMA	107	97	100	97	106	57	172	129	117
IMAR^2+TMMA	102	92	99	100	104	63	128	116	106
CIMAR+TMMA	95	81	101	100	100	56	80	92	99
NIMAR+TMMA	94	84	100	97	99	54	82	91	96
SMA+TMMA	94	82	101	97	99	55	82	92	96
DyRMMA+TMMA	93	94	97	93	91	71	90	96	97
LBMA+LMMA	106	99	100	98	106	63	168	131	116
IMAR^2+LMMA	104	92	100	98	103	56	130	118	105
CIMAR+LMMA	95	78	100	98	99	57	81	92	95
NIMAR+LMMA	93	78	101	101	99	54	82	89	96
SMA+LMMA	93	80	100	97	100	56	81	91	95
DyRMMA+LMMA	95	86	97	91	90	79	90	99	94
RMMA	190	200	102	105	115	223	157	223	422
TMMA	101	94	101	99	102	87	94	96	97
LMMA	100	97	100	97	100	88	95	89	101

Table 4.7: Normalised execution times (%) for Experiment Single in Server ctnuma1. Lower is better.

rollback phase. For the DyRMMA, the entire algorithm depends on equation (3.23), so it is clear that, if the calculation of performance is not accurate, the strategy will not be accurate either. Despite the aforementioned problems, these algorithms still manage to slightly improve the Baseline in most of the cases, particularly DyRMMA, which reduces the execution times for every benchmark in the experiment.

On the positive side, three algorithms should be highlighted: CIMAR, NIMAR and SMA. These strategies are capable of improving thread location by placing them in their preferred nodes, as shown in Figure 4.8. Thus, local operations are enforced which allows for better execution times. Take, for example, the LU.C benchmark. According to the obtained measurements, threads are in their preferred node 45 % of the time with the Baseline. This number increases up to 83 % when using CIMAR algorithm, up to 92 % for NIMAR and up to 85 % for SMA.

Regarding memory migration algorithms, the random strategy—RMMA—shows

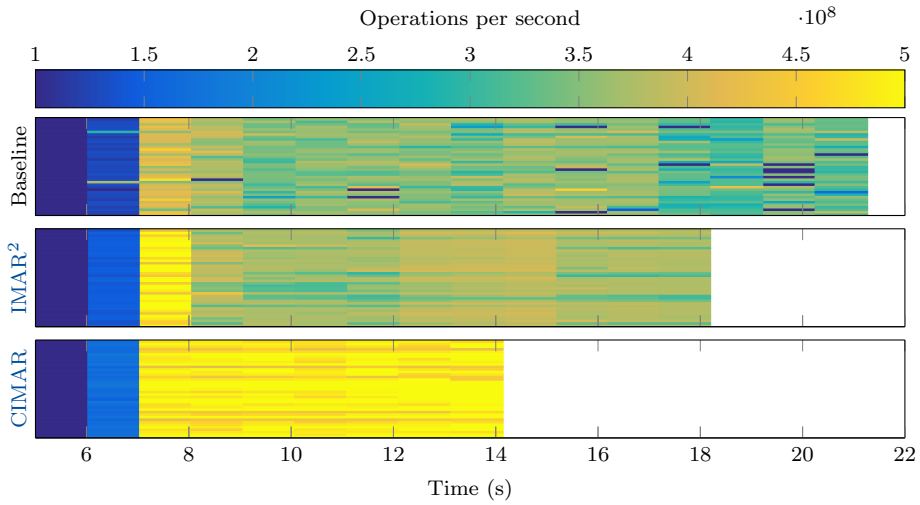


Figure 4.7: Traces showing operations per second for CG.C benchmark running under Baseline, IMAR^2 and CIMAR algorithms in Experiment Single in Server ctnuma1. Higher is better.

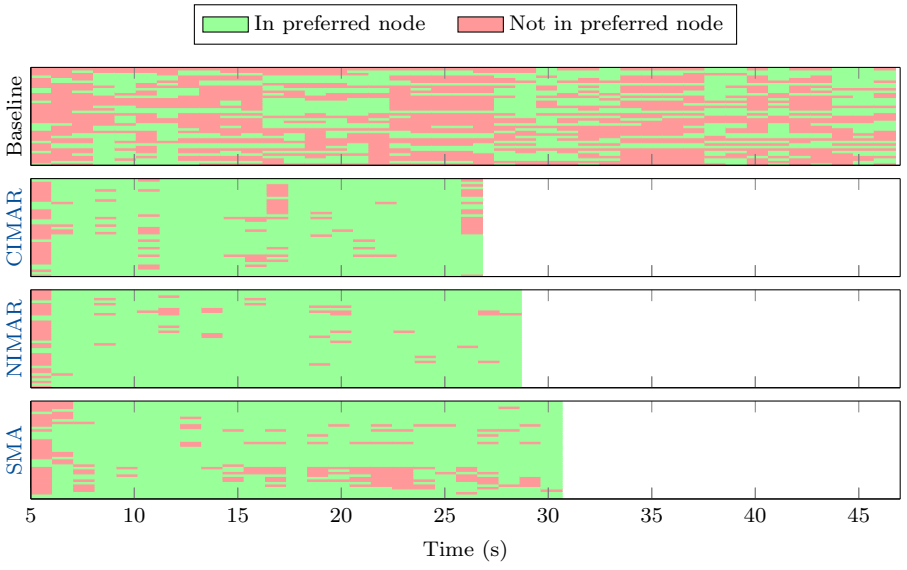


Figure 4.8: Traces showing whether threads are in their preferred node or not for LU.C benchmark running under Baseline, CIMAR, NIMAR and SMA algorithms in Experiment Single in Server ctnuma1.

the possible impact of bad decisions when migrating memory pages, with performance being heavily downgraded. On the other hand, TMMA and LMMA reduce execution times slightly. The issues regarding memory pages migrations are further discussed in Section 4.9. Examples of traces on how each algorithm affects execution time are shown in Figure 4.9.

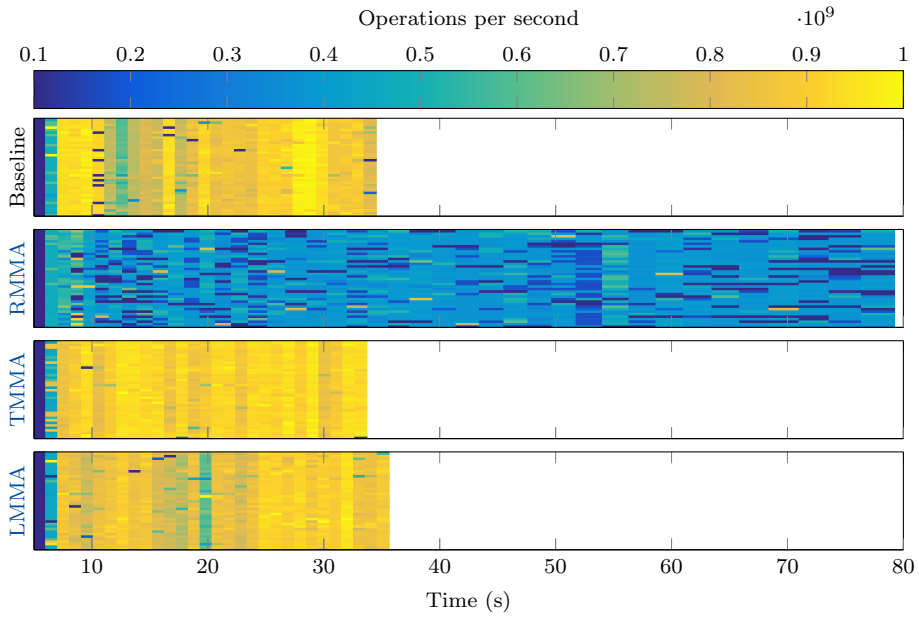


Figure 4.9: Traces showing operations per second for SP.C benchmark running under Baseline, RMMA, TMMA, and LMMA algorithms in Experiment Single in Server ctnuma1. Higher is better.

Finally, it should be noted that when memory migrations are used alongside thread migration algorithms, little effect on performance can be observed since thread placement plays the leading role.

4.5.2 Server ctnuma2

Normalised execution times are shown in Table 4.8, while raw execution times can be consulted in Table B.2 in Appendix B.

In Server ctnuma2 the interleaved mapping achieves big improvements, reducing execution times up to 65 %. The benchmarks, when executed in this particular server, benefit from the increased bandwidth caused by the Interleaved mapping, as illustrated

Algorithm	BT.C	CG.C	EP.D	FT.C	IS.D	LU.C	MG.D	SP.C	UA.C
Baseline	100	100	100	100	100	100	100	100	100
Direct	264	262	100	252	442	141	626	528	248
Interleave	72	113	101	48	97	35	116	83	84
CRA	83	102	105	81	101	65	104	89	95
LBMA	85	108	102	78	102	89	98	88	89
IMAR^2	84	103	99	88	103	57	98	93	89
CIMAR	80	107	101	81	101	51	98	89	86
NIMAR	82	101	101	81	104	72	98	89	88
SMA	89	102	101	76	101	63	99	90	86
DyRMMA	81	119	100	83	102	57	109	91	96
LBMA+TMMA	84	100	100	81	101	60	100	89	91
IMAR^2+TMMA	80	102	101	87	103	58	98	86	91
CIMAR+TMMA	89	109	101	87	102	69	105	90	89
NIMAR+TMMA	77	101	101	71	101	65	102	87	90
SMA+TMMA	87	106	101	82	100	57	99	93	89
DyRMMA+TMMA	103	100	101	92	99	66	116	91	95
LBMA+LMMA	84	98	101	84	102	62	99	88	91
IMAR^2+LMMA	85	104	103	79	103	62	104	91	89
CIMAR+LMMA	90	102	102	79	100	62	108	88	90
NIMAR+LMMA	80	96	102	92	102	75	100	89	90
SMA+LMMA	85	102	102	85	101	77	100	95	90
DyRMMA+LMMA	83	99	101	80	101	81	113	88	96
RMMA	129	103	100	89	102	186	108	139	183
TMMA	98	105	106	92	102	97	103	101	101
LMMA	88	105	102	88	101	102	106	94	107

Table 4.8: Normalised execution times (%) for Experiment Single in Server ctnuma2. Lower is better.

in Figure 4.10. Note that the Direct mapping is far from the Baseline given its poor use of both locality and bandwidth.

Regarding the algorithms implemented in Thanos, almost every algorithm improves the Baseline, even CRA which places the threads completely randomly across the server or LBMA which relies heavily on randomness in the decision-making process. This is due to the fact that CRA pins the threads to particular cores, whose cache memory is kept in a "hot" state until the threads are assigned another CPU. Following the same reasoning, IMAR^2 and CIMAR achieve the best results. Those algorithms which map threads to nodes—NIMAR, SMA and DyRMMA—have slightly worse performance but still improve Baseline significantly in most of the benchmarks. Figure 4.11 shows that even when NIMAR places the threads in their preferred node more often, CIMAR achieves a better execution time because of the more efficient use of the cache.

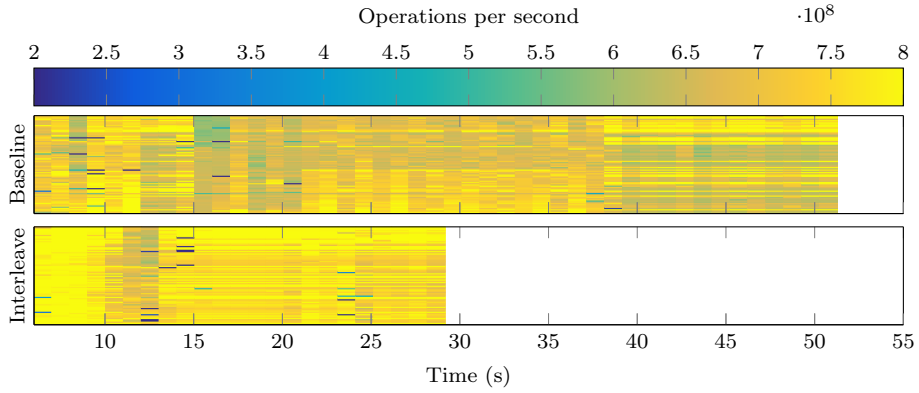


Figure 4.10: Traces showing operations per second for LU.C benchmark running under Baseline and Interleave mapping in Experiment Single in Server ctnuma2. Higher is better.

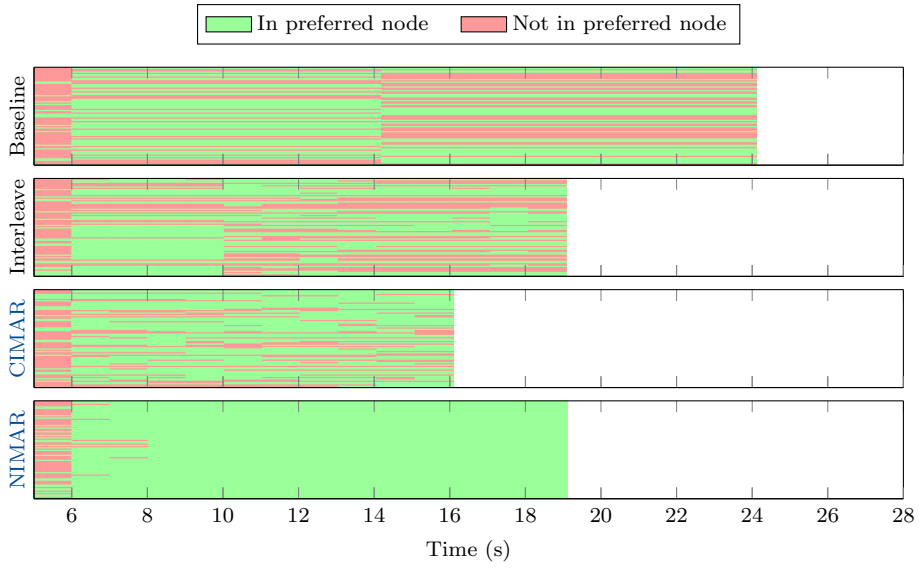


Figure 4.11: Traces showing whether threads are in their preferred node or not for SP.C benchmark running under Baseline, Interleave, CIMAR and NIMAR algorithms in Experiment Single in Server ctnuma2.

Finally, it should be noted that even memory migration algorithms improve the execution times, particularly LMMA. Reducing the distance between the threads and the most frequently used memory pages reduces the average latency of memory operations from 11 cycles with the Baseline to 9.41 cycles with LMMA. Furthermore, the standard deviation of the latency is greatly reduced, from 51.31 to 19.70. Thanks to the reduction of the latency, the CPU usage is higher, as shown in Figure 4.12, and execution times are shortened.

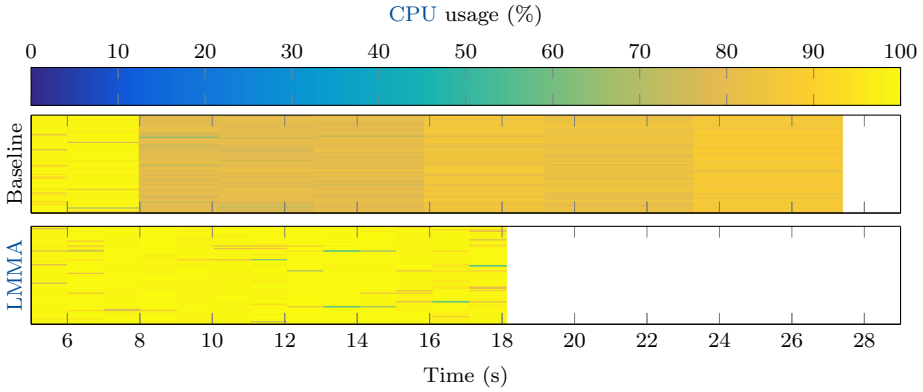


Figure 4.12: Traces showing CPU usage for BT.C benchmark running under Baseline and LMMA algorithm in Experiment Single in Server ctnuma2. Higher is better.

4.6 Experiment Interactive

In this experiment, small tasks are launched at predetermined times, emulating an interactive server, where the impact of migrations is expected to be higher than in Section 4.5. Note that Tables 4.9 and 4.10 include the normalised total execution time of the experiment, which is the time between the start of the first task and the end of the last task. Additionally, the normalised accumulated execution time of the tasks is shown in the last column of the tables, which is the sum of the execution times of all the tasks.

4.6.1 Server ctnuma1

Table 4.9 shows the normalised execution times for the Experiment Interactive executed in Server ctnuma1. Raw execution times are shown in Table B.3 in Appendix B. Normalised computation times are shown in Figure 4.13.

The trends seen in Section 4.5 are still valid in the Experiment Interactive regarding the algorithms implemented in Thanos. The Direct mapping has the best performance

Algorithm	BT.C	CG.C	LU.C	SP.C	Total	Accum.
Baseline	100.0	100.0	100.0	100.0	100.0	100.0
Direct	95.7	55.1	89.7	91.5	98.3	85.3
Interleave	106.3	116.3	105.7	119.4	103.9	110.8
CRA	116.7	125.3	116.9	133.1	108.9	121.8
LBMA	122.7	129.5	121.7	134.9	109.7	126.3
IMAR^2	105.7	99.2	105.1	108.8	102.2	105.5
CIMAR	96.0	57.6	99.6	83.7	96.6	86.9
NIMAR	98.6	66.6	101.8	90.5	97.3	91.2
SMA	97.2	58.6	102.7	84.3	97.3	87.8
DyRMMA	107.5	114.3	108.0	122.0	105.6	111.8
LBMA+TMMA	122.3	120.4	122.8	133.4	108.9	124.6
IMAR^2+TMMA	105.5	90.3	106.9	109.2	101.7	103.2
CIMAR+TMMA	96.6	57.9	101.2	84.4	97.6	87.5
NIMAR+TMMA	98.8	59.8	102.8	89.5	97.8	89.3
SMA+TMMA	98.1	57.9	102.8	88.5	97.1	88.9
DyRMMA+TMMA	107.5	111.2	107.5	121.9	105.6	111.1
LBMA+LMMA	120.2	127.8	123.9	133.9	109.2	125.0
IMAR^2+LMMA	103.9	91.3	105.1	110.7	103.6	104.3
CIMAR+LMMA	96.8	59.3	101.2	84.4	97.1	87.3
NIMAR+LMMA	98.4	62.0	103.1	86.2	97.3	89.7
SMA+LMMA	98.3	57.9	102.5	84.8	97.3	87.7
DyRMMA+LMMA	107.1	112.3	107.9	122.7	105.3	111.4
RMMA	123.7	118.6	144.2	147.6	110.6	132.4
TMMA	101.5	79.8	99.8	97.9	100.0	97.0
LMMA	101.0	96.1	100.5	103.2	100.7	100.1

Table 4.9: Normalised execution times (%) for Experiment Interactive in Server ctnuma1. Lower is better.

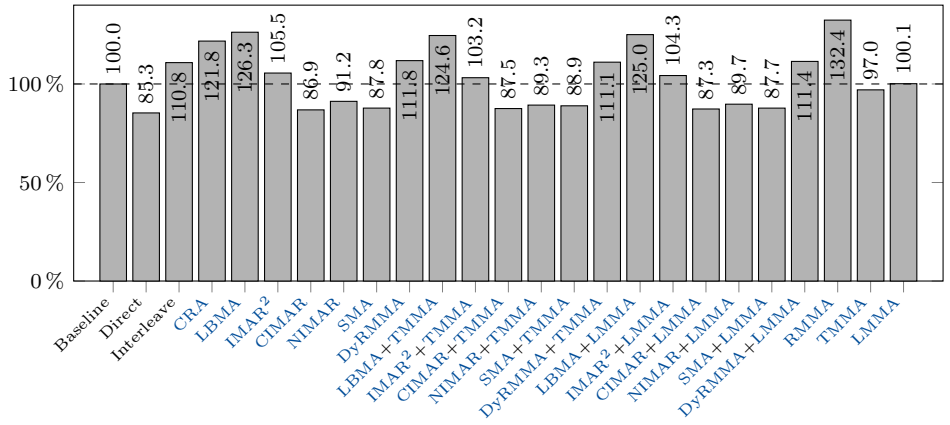


Figure 4.13: Normalised computation time for Experiment Interactive in Server ctnuma1. Lower is better.

in most cases. Note that it is the only mapping which improves significantly the Baseline time for the LU benchmark, this is due to the lower average latency of memory operations, as shown in Figure 4.14. Since the threads are located in the same node, cache-coherency protocols work more efficiently, and better use of the cache can be done. This is not the case for the Baseline, nor other strategies as shown in Figure 4.15.

CIMAR, NIMAR and SMA are the only thread migration strategies that can improve the Baseline execution times. Should be highlighted the improvements in the CG.C benchmark, with reduction of execution times between 34 % and 43 %, and the SP.C, whose performance has been increased between 10 % and 17 %.

4.6.2 Server ctnuma2

Table 4.10 shows the normalised execution times for the Experiment Interactive executed in Server ctnuma2. Raw execution times are shown in Table B.4 in Appendix B. Normalised computation times are shown in Figure 4.16.

Results show significant performance losses in the Direct mapping, particularly for the LU.C benchmark. The reason for this behaviour is the same as mentioned in Section 4.5.2. The Interleave mapping is not as good as in the Experiment Single, achieving results similar to the Baseline.

Regarding the algorithms implemented in Thanos, several thread migration algorithms improve the Baseline, particularly IMAR^2, CIMAR and SMA. These strategies improve slightly the execution times of the SP.C benchmark, up to 6 %. Nevertheless, it is in the CG.C benchmark where they perform the best, increasing performance

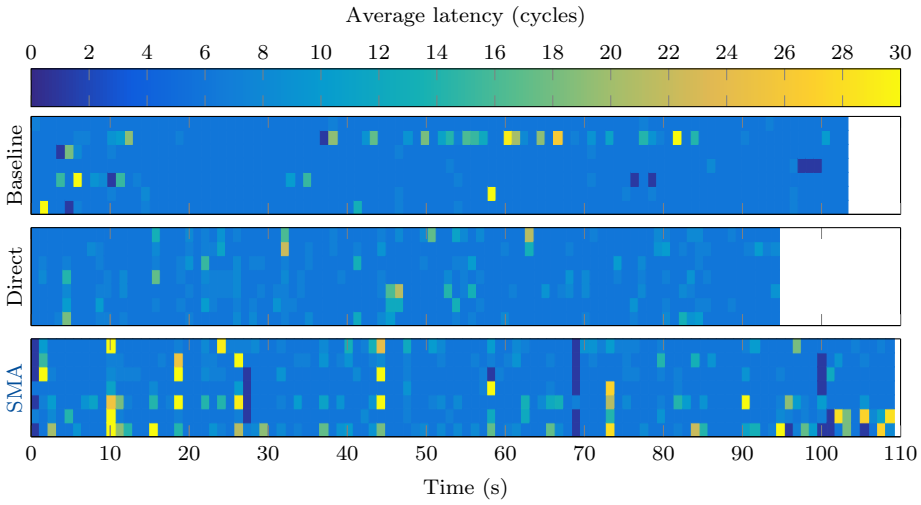


Figure 4.14: Traces showing average latency of memory operations for LU.C benchmark running under Baseline, Direct, and SMA algorithms in Experiment Interactive in Server ctnuma1. Lower is better.

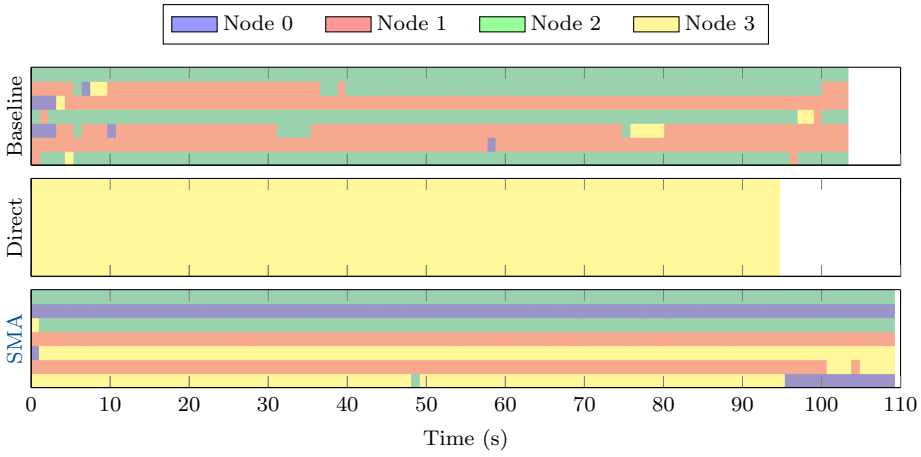


Figure 4.15: Traces showing thread location for LU.C benchmark running under Baseline, Direct, and SMA algorithms in Experiment Queue.

Algorithm	BT.C	CG.C	LU.C	SP.C	Total	Accum.
Baseline	100.0	100.0	100.0	100.0	100.0	100.0
Direct	184.9	121.9	2,837.5	243.8	705.9	895.4
Interleave	98.3	110.5	90.8	101.0	99.0	99.7
CRA	108.6	130.8	107.9	113.6	102.5	113.1
LBMA	105.5	86.3	112.6	105.4	100.5	106.2
IMAR^2	98.6	78.3	101.5	97.7	99.0	96.5
CIMAR	97.5	75.9	101.7	94.0	99.0	96.6
NIMAR	104.0	81.8	99.8	102.7	99.5	99.7
SMA	100.5	79.3	107.0	95.1	97.5	98.1
DyRMMA	106.4	127.2	100.4	115.1	102.9	110.6
LBMA+TMMA	109.9	83.9	122.3	106.6	101.0	106.5
IMAR^2+TMMA	98.5	76.6	106.7	96.6	97.5	97.8
CIMAR+TMMA	104.5	80.6	106.7	98.7	98.5	101.9
NIMAR+TMMA	102.7	91.5	99.9	102.8	99.5	99.7
SMA+TMMA	99.6	76.6	101.5	91.4	96.6	95.2
DyRMMA+TMMA	109.4	129.7	106.5	115.8	102.5	114.8
LBMA+LMMA	104.2	81.1	106.5	104.5	100.5	101.5
IMAR^2+LMMA	99.9	82.9	109.5	97.9	98.5	101.9
CIMAR+LMMA	98.2	75.4	101.3	96.8	99.0	95.1
NIMAR+LMMA	101.2	90.0	99.1	98.5	98.0	98.8
SMA+LMMA	100.3	72.2	97.3	90.5	96.6	94.0
DyRMMA+LMMA	106.8	128.3	104.3	113.7	102.5	111.6
RMMA	115.9	128.4	154.4	143.8	107.8	137.1
TMMA	99.9	101.2	101.8	102.2	100.5	102.8
LMMA	101.0	96.9	101.3	98.3	99.5	100.1

Table 4.10: Normalised execution times (%) for Experiment Interactive in Server ctnuma2. Lower is better.

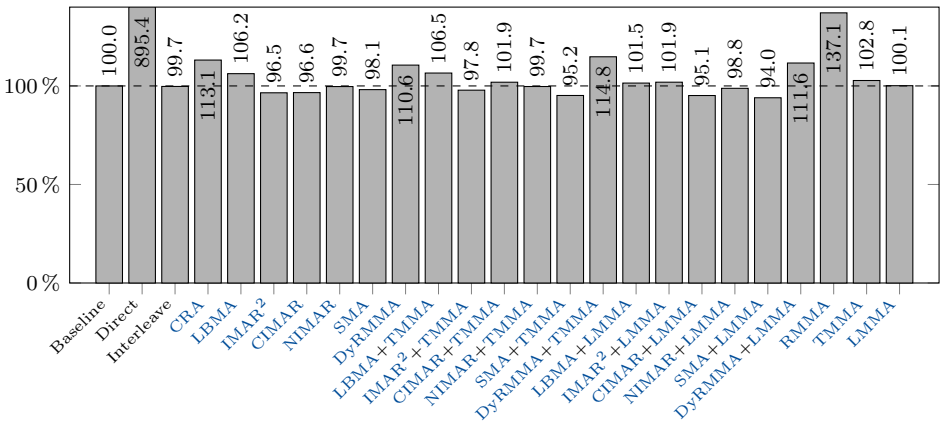


Figure 4.16: Normalised computation time for Experiment Interactive in Server ctnuma2. Lower is better.

by up to 24 %. This is caused because Thanos algorithms map the threads into their preferred node a higher fraction of the time compared to the Baseline. For CIMAR, it is a 91 % of the time, against the 85 % of the baseline, see Figure 4.17.

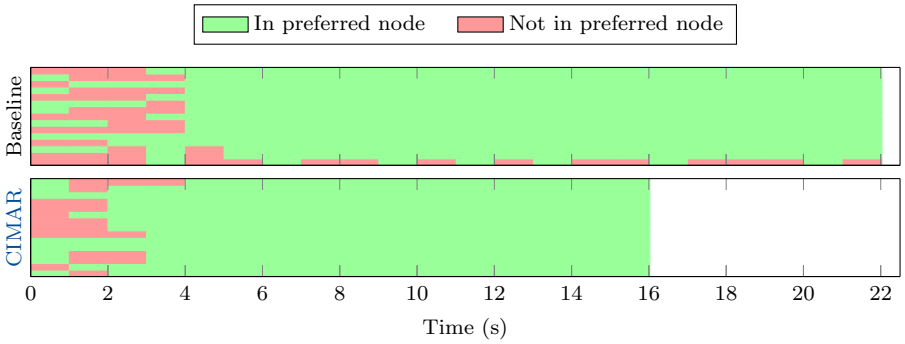


Figure 4.17: Traces showing whether threads are in their preferred node or not for CG.C benchmark running under Baseline, and CIMAR algorithms in Experiment Interactive in Server ctnuma2.

Finally, it should be noted that the memory migration algorithms have a very slight impact, following the tendency shown in Section 4.5.2.

4.7 Experiment Queue

By simulating several users, the server is kept busy with a collection of different tasks. Thus, it is expected that the impact of migration is the highest in this highly dynamic scenario.

Note that Tables 4.11 and 4.12 show the total and the accumulated—normalised—execution times of the benchmarks, which have the same meaning as in Section 4.6.

4.7.1 Server ctnuma1

Table 4.11 shows the normalised execution times for Experiment Queue. Raw execution times are shown in Table B.5 in Appendix B. Furthermore, a graphic overview of these results is shown in Figure 4.18.

Algorithm	BT.C	CG.C	EP.D	FT.C	IS.D	LU.C	MG.D	SP.C	UA.C	Total	Accum.
Baseline	100	100	100	100	100	100	100	100	100	100	100
Direct	91	58	99	93	60	86	83	80	90	85	86
Interleave	116	162	101	112	161	117	149	143	129	130	129
CRA	103	104	100	100	122	106	107	100	102	106	106
LBMA	100	103	99	98	111	109	100	99	107	106	103
IMAR^2	101	109	99	99	112	106	97	99	115	105	103
CIMAR	99	85	100	99	105	97	77	86	88	96	93
NIMAR	95	71	99	97	105	99	71	76	85	89	88
SMA	95	69	99	98	106	97	69	74	85	89	88
DyRMMA	107	140	99	111	139	110	134	117	112	117	117
LBMA+TMMA	101	99	100	101	110	105	102	99	112	106	103
IMAR^2+TMMA	99	101	105	101	114	100	100	94	99	105	102
CIMAR+TMMA	100	82	101	100	104	98	74	82	91	97	91
NIMAR+TMMA	95	70	99	96	102	99	72	76	87	90	89
SMA+TMMA	95	69	99	96	107	98	70	75	87	89	88
DyRMMA+TMMA	107	134	100	112	138	111	137	117	113	118	118
LBMA+LMMA	96	100	108	101	113	100	99	98	106	106	103
IMAR^2+LMMA	100	98	99	104	118	106	105	99	102	105	104
CIMAR+LMMA	98	69	100	98	104	96	69	78	84	91	87
NIMAR+LMMA	95	70	99	96	108	99	72	75	87	90	89
SMA+LMMA	96	68	99	95	107	99	66	75	86	89	88
DyRMMA+LMMA	107	132	99	113	137	110	135	116	112	116	117
RMMA	133	136	100	111	109	129	113	161	137	121	122
TMMA	98	81	99	99	103	98	99	98	102	100	99
LMMA	99	92	99	101	100	97	98	95	103	99	99

Table 4.11: Normalised execution times for Experiment Queue in Server ctnuma1. Lower is better.

The Direct mapping is still the best placement policy, but the Baseline performs better than in previous experiments. Again, the Interleaved mapping offers poor results for this particular system. As shown in Figure 4.19, when running the CG.C benchmark with the Interleave mapping, the threads are in their preferred node only 17 % of the time. With the Baseline, this time increases up to 84 %, and it is a perfect

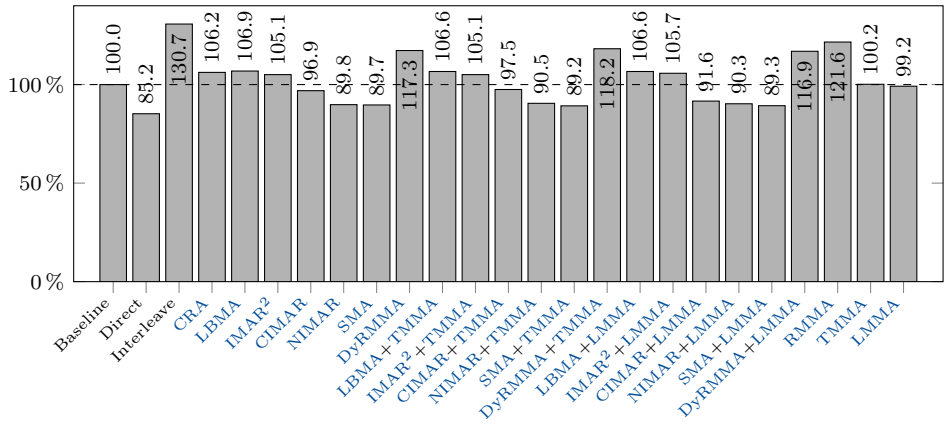


Figure 4.18: Normalised total execution time for Experiment Queue in Server ctnuma1. Lower is better.

100 % with the Direct mapping—the expected outcome. Note that the Direct mapping could cause memory congestion, as mentioned in Section 1.3.1. It was not the case in Server ctnuma1, but it is a phenomenon to be aware of when using this option.

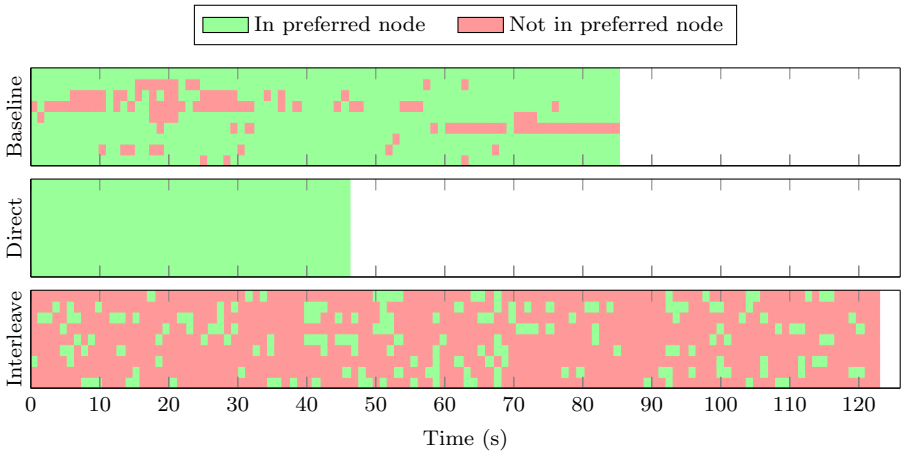


Figure 4.19: Traces showing whether threads are in their preferred node or not for CG.C benchmark running under Baseline, Direct and Interleave algorithms in Experiment Queue in Server ctnuma1.

Among the thread placement algorithms implemented in Thanos, the best-performing

strategies are CIMAR, NIMAR and SMA. On the other hand, LBMA, IMAR^2 and DyRMMA cannot improve the baseline. Taking the LU.C benchmark as an example, see Figure 4.20, the Baseline manages to have the threads in their preferred node 79 % of the time. For LBMA and DyRMMA, this number is reduced up to 50 % approximately. Despite that IMAR^2 keeps the threads in their preferred node about 60 % of the time, it is not enough to improve the execution times.

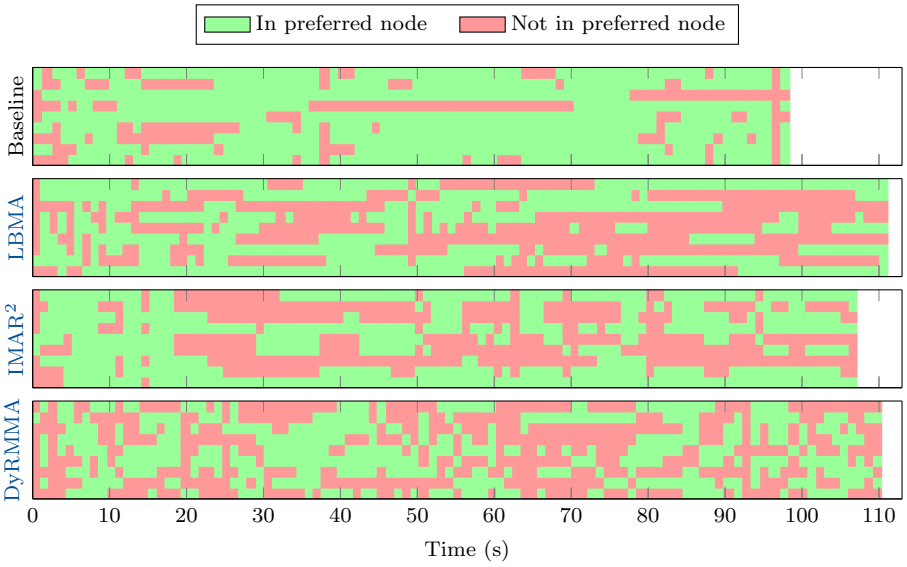


Figure 4.20: Traces showing whether threads are in their preferred node or not for LU.C benchmark running under Baseline, LBMA, IMAR^2 and DyRMMA algorithms in Experiment Queue in Server ctnuma1.

There are cases where the Direct mapping is not the best placement strategy as is the case for the SP.C benchmark. NIMAR and SMA obtain better results than the Direct by placing the threads across different nodes as shown in Figure 4.21. The threads are still in their preferred node—see Figure 4.22—but the resources of the system are used in a better way. Thus, operations per second performed are increased, as shown in Figure 4.23, and execution times are improved.

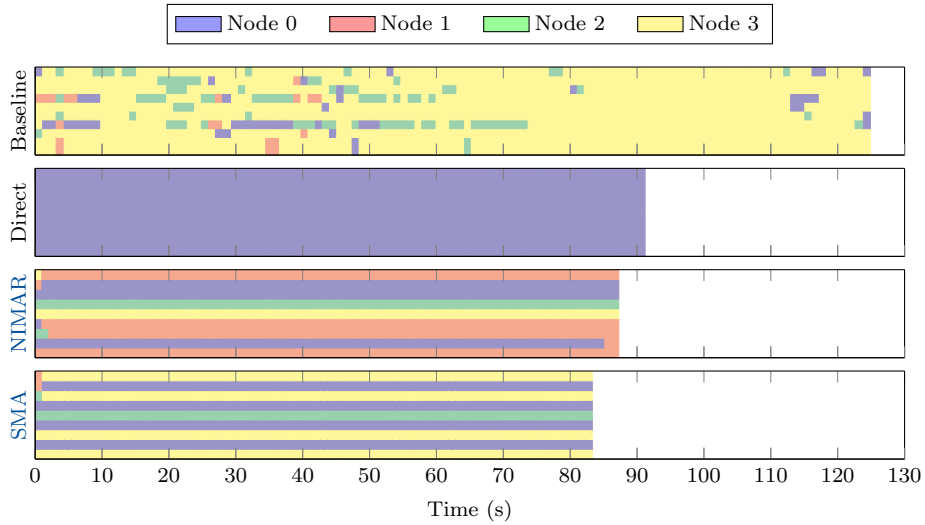


Figure 4.21: Traces showing node in which threads are running for SP.C benchmark running under Baseline, Direct, NIMAR and SMA algorithms in Experiment Queue in Server ctnuma1.

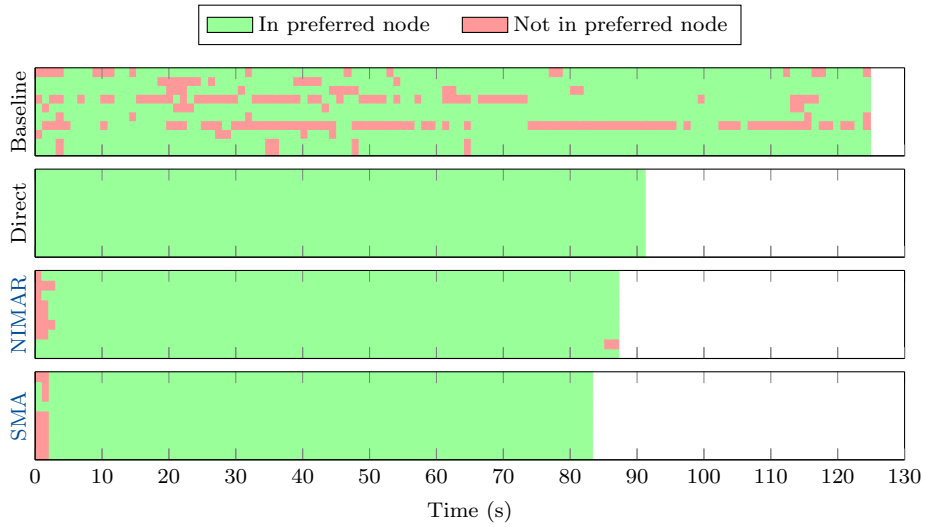


Figure 4.22: Traces showing whether threads are in their preferred node or not for SP.C benchmark running under Baseline, Direct, NIMAR and SMA algorithms in Experiment Queue in Server ctnuma1.

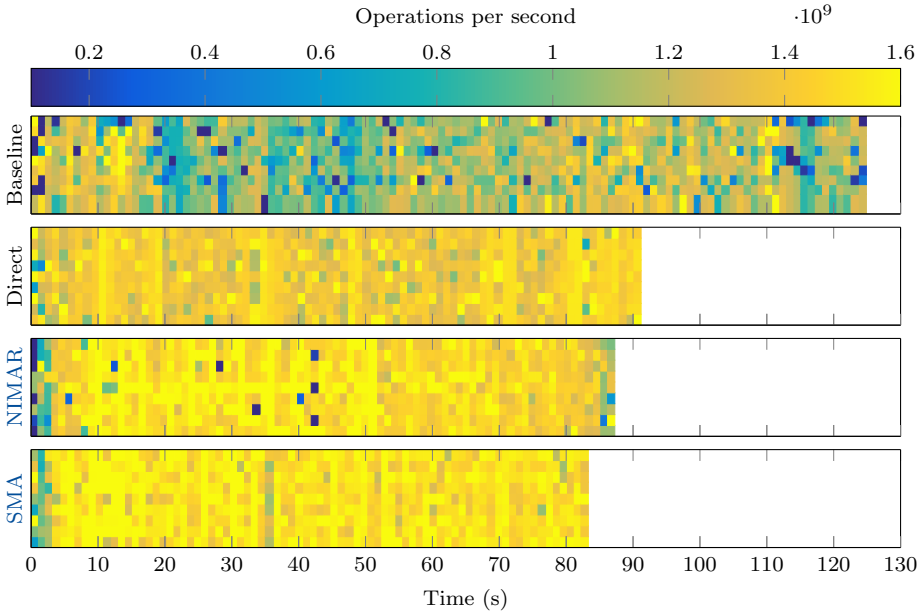


Figure 4.23: Traces showing operations per second for SP.C benchmark running under Baseline, Direct, NIMAR and SMA algorithms in Experiment Queue in Server ctnuma1. Higher is better.

4.7.2 Server ctnuma2

Table 4.12 shows the normalised execution times for Server ctnuma2. Raw times are shown in Table B.6 in Appendix B, and a graphical overview is shown in Figure 4.24.

In this experiment, the Direct mapping improves the Baseline by 10 %, while the Interleave increase execution times by 17 %. Since the system has several programs running simultaneously, the potential increase of bandwidth of the Interleave mapping is divided among the programs running. That makes the latency of memory the leading factor, as shown in Figure 4.25. Thus, the Direct is the most adequate mapping.

The algorithms implemented in Thanos tend to slightly improve the execution times compared to the Baseline. The only ones that could not increase the performance are CRA, IMAR^2 and RMMA. It is expected that CRA and RMMA have a negative impact on performance since they are random algorithms, but it is not the case for IMAR^2.

Figure 4.26 shows that CIMAR improves the execution times because of the typically higher number of operations per second. Since threads are pinned to particular cores, better use of the cache is enforced, improving performance. Despite that IMAR^2

Algorithm	BT.C	CG.C	EP.D	FT.C	IS.D	LU.C	MG.D	SP.C	UA.C	Total	Accum.
Baseline	100	100	100	100	100	100	100	100	100	100	100
Direct	89	56	98	86	64	83	94	89	88	90	88
Interleave	113	146	103	108	134	113	118	125	119	117	118
CRA	105	109	99	97	114	106	100	105	103	105	103
LBMA	95	90	104	91	102	100	85	96	96	97	94
IMAR^2	112	122	99	100	125	111	110	114	118	109	112
CIMAR	99	83	100	89	109	104	84	85	103	93	95
NIMAR	97	78	98	90	103	95	80	89	95	93	90
SMA	101	92	100	93	105	102	88	97	99	98	96
DyRMMA	100	83	99	89	105	106	85	91	102	93	95
LBMA+TMMA	96	80	97	89	100	98	77	89	95	89	90
IMAR^2+TMMA	113	114	99	99	126	112	111	116	121	110	113
CIMAR+TMMA	101	84	97	88	105	101	84	87	104	93	94
NIMAR+TMMA	98	80	99	87	99	101	80	89	96	91	91
SMA+TMMA	102	98	101	95	105	100	94	99	102	101	99
DyRMMA+TMMA	101	82	99	91	104	102	83	87	101	93	93
LBMA+LMMA	97	77	100	91	102	99	79	90	95	91	91
IMAR^2+LMMA	113	118	100	98	125	112	111	115	121	109	112
CIMAR+LMMA	101	81	97	90	107	103	84	88	103	93	94
NIMAR+LMMA	95	78	100	89	99	101	82	88	96	90	91
SMA+LMMA	101	89	102	99	105	101	93	97	100	98	98
DyRMMA+LMMA	101	79	99	88	104	102	83	88	103	92	94
RMMA	110	103	98	100	108	111	105	111	107	105	107
TMMA	100	89	100	100	104	102	100	98	99	99	99
LMMA	102	92	98	102	106	101	97	98	98	100	99

Table 4.12: Normalised execution times for Experiment Queue in Server ctnuma2. Lower is better.

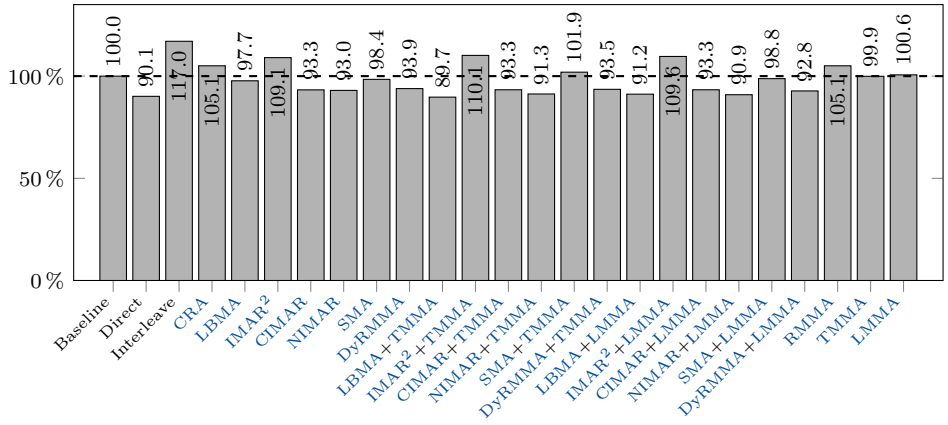


Figure 4.24: Normalised total execution time for Experiment Queue in Server ctnuma2. Lower is better.

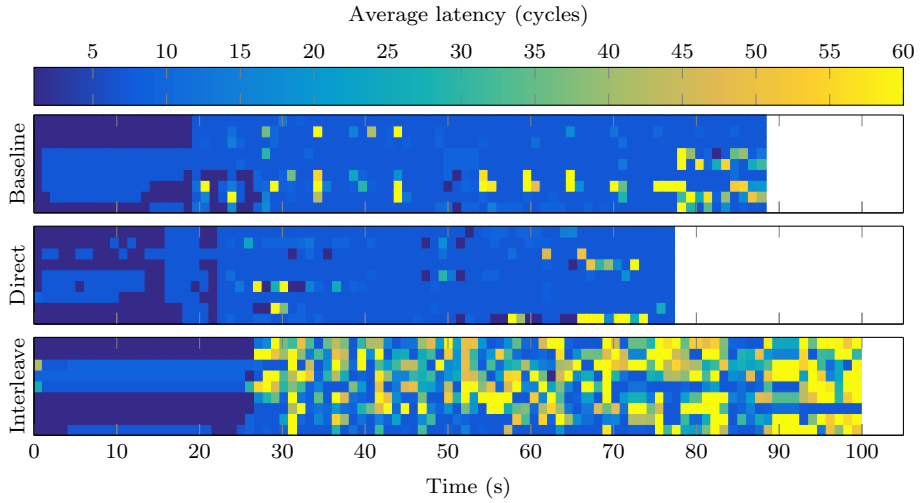


Figure 4.25: Traces showing average latency of memory operations for IS.D benchmark running under Baseline, Direct, and Interleave mappings in Experiment Queue in Server ctnuma2. Higher is better.

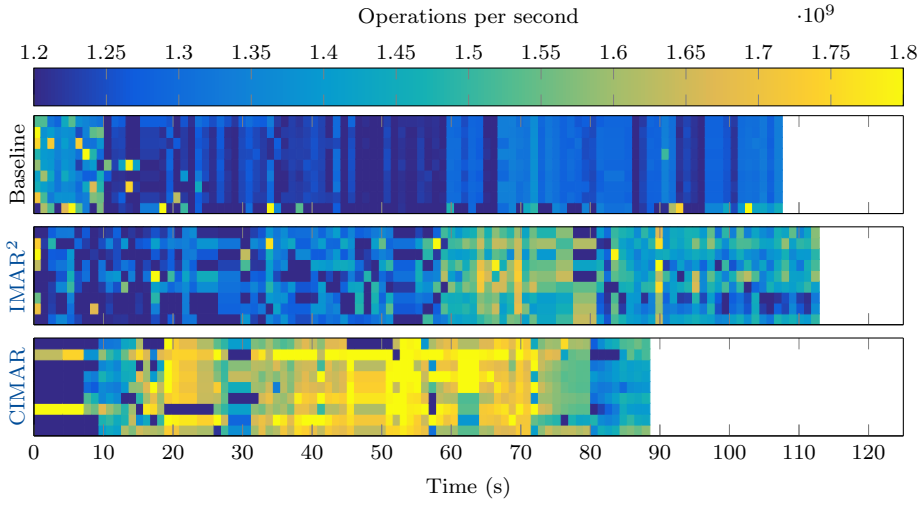


Figure 4.26: Traces showing operations per second for SP.C benchmark running under Baseline, IMAR^2, and CIMAR algorithms in Experiment Queue in Server ctnuma2. Higher is better.

does the same kind of pinning, it is not capable of reducing the execution times because of the uneven performance amongst the threads of the process, particularly in the early stages of the program.

Memory migration algorithms have a slight impact on performance. The CG.C is the only benchmark whose execution times are significantly improved when using these algorithms. Metrics show that with the Baseline, the threads spend the 88 % of the time in their preferred node. This number is increased up to 96 % with TMMA, see Figure 4.27. Though, LMMA improves Baseline execution times because of a slightly better use of the CPU, which is kept busy more often.

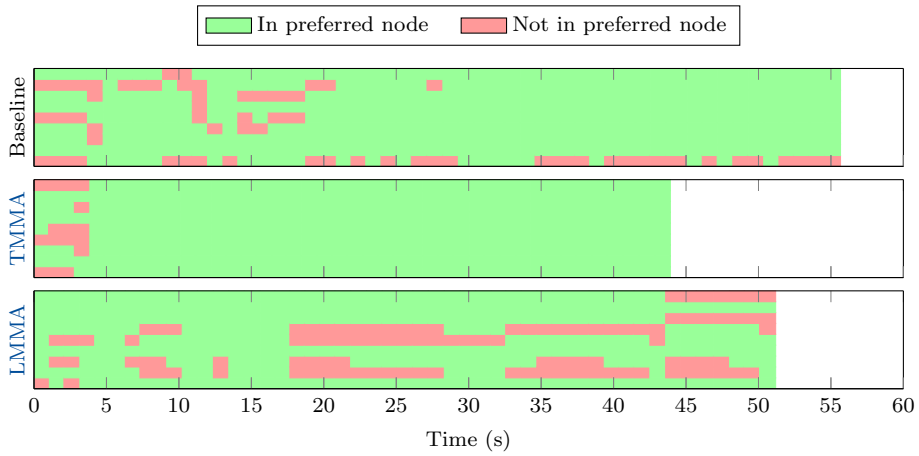


Figure 4.27: Traces showing whether threads are in their preferred node or not for CG.D benchmark running under Baseline, TMMA and LMMA algorithms in Experiment Queue in Server ctnuma2.

4.8 Energy

Figures 4.28 and 4.29 show a comparison between execution time and energy and the linear regression derived from the data. It can be seen that, in this kind of server, energy is almost linear with execution times. Note that the slope of the regression is similar for every experiment, reinforcing the argument that there is a linear relationship between both metrics.

For memory and CPU related workload, with negligible use of I/O operations, the power usage usually is at its maximum all the time. As mentioned in Section 1.1.2, it is the efficiency of memory operations and CPU instructions that might reduce the energy requirements. On one hand, the efficiency of CPU instructions can be improved in the binary generation process, which is out of the scope of this work. On the other

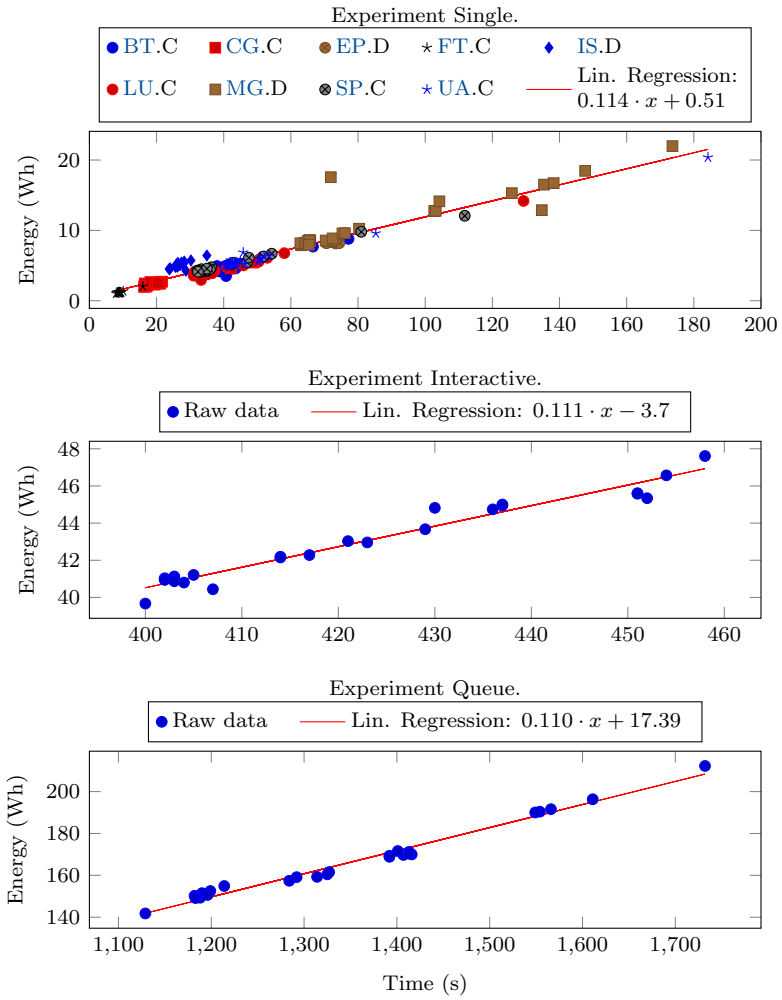


Figure 4.28: Energy (Wh) against time (s) for the results obtained in Experiments Single, Interactive and Queue in Server ctnuma1 and the derived linear regressions.

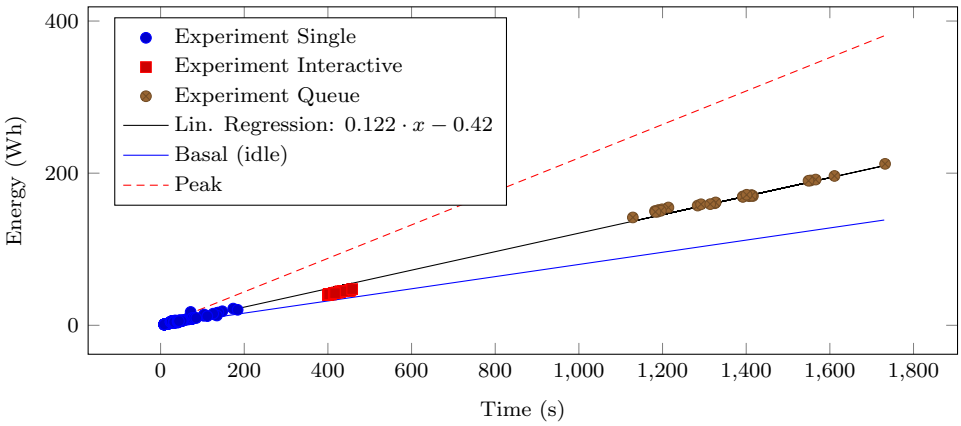


Figure 4.29: Energy (Wh) against time (s) for the results obtained in Experiments Single, Interactive and Queue in Server ctnuma1 and the combined linear regression.

hand, memory operations can be optimised by the scheduler, which is one of the goals of this thesis.

Figure 4.30 shows the total energy consumption for the Experiments Single, Interactive and Queue in Server ctnuma1. For Experiments Interactive and Queue, it is not possible to know the energy consumption of each particular task, only the total energy consumption. For Experiment Single, it is possible to assume that the power demands of other programs like daemons or OS processes are negligible, so the energy consumption of each task is detailed in Table 4.13.

Note that some points regarding Experiment Interactive are slightly under the linear regression. In this experiment, not all resources are used all the time, so some CPUs can enter a low-power state, reducing energy consumption.

Those algorithms that improved the Baseline execution time in each experiment did roughly the same regarding the energy. A good example is the reduction of the energy consumption for the LU benchmark in Experiment Single, where those strategies that improved the most the execution time also reduced the energy consumption. Algorithms like SMA, CIMAR and NIMAR reduced the energy around 40 %. In situations where the reduction in execution times is not so big, the energy consumption is still reduced proportionally. This can be extended to the other experiments, where the energy consumption is reduced up to 12 %. Thus, it can be stated that the Direct mapping, SMA, CIMAR and NIMAR algorithms are the best ones in terms of energetic efficiency.

Summarising, the most important factor related to energy efficiency in the context of HPC systems is performance. The sooner the tasks are completed, the sooner the processors can enter an idle state to reduce their power usage.

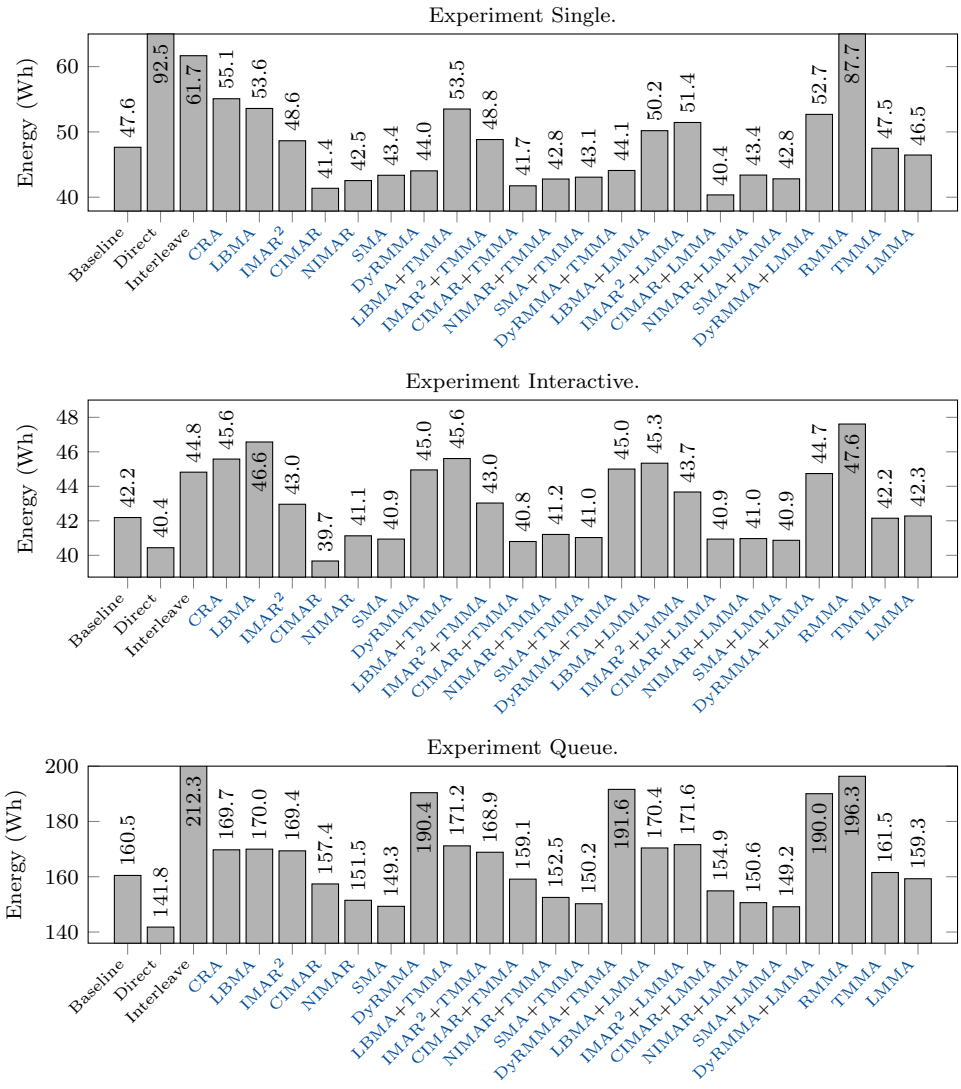


Figure 4.30: Energy (Wh) for Experiments Single, Interactive and Queue in Server ctnuma1. Lower is better.

Algorithm	BT.C	CG.C	EP.D	FT.C	IS.D	LU.C	MG.D	SP.C	UA.C	Total
Baseline	3.51	2.41	8.75	1.17	5.38	6.76	10.23	4.76	5.48	47.65
Direct	7.69	5.42	8.13	2.04	6.43	5.55	36.19	12.07	9.58	92.50
Interleave	5.28	2.70	8.62	1.35	5.57	3.93	21.98	6.69	6.27	61.67
CRA	5.27	2.61	8.28	1.09	4.27	3.77	18.46	6.29	6.38	55.08
LBMA	5.16	2.64	8.42	1.16	5.16	4.26	16.50	5.48	5.54	53.59
IMAR^2	4.96	2.34	8.31	1.21	5.13	3.88	12.75	5.17	5.50	48.64
CIMAR	4.29	1.97	8.56	1.22	4.80	3.89	7.89	4.25	5.09	41.37
NIMAR	4.31	2.16	8.70	1.16	5.12	3.90	8.10	4.29	5.30	42.54
SMA	4.48	2.25	8.66	1.21	5.04	4.21	8.17	4.25	5.49	43.36
DyRMMA	4.45	2.52	8.19	1.01	4.62	6.10	8.56	4.21	4.99	44.04
LBMA+TMMA	4.59	2.47	8.30	1.14	5.34	3.85	16.71	5.46	5.95	53.51
IMAR^2+TMMA	4.85	2.34	8.49	1.17	5.41	4.40	12.78	5.17	5.25	48.83
CIMAR+TMMA	4.16	2.02	8.62	1.19	4.91	3.71	8.26	4.42	5.40	41.75
NIMAR+TMMA	4.73	2.35	8.60	1.17	5.11	3.56	8.61	4.36	5.46	42.79
SMA+TMMA	4.58	2.25	8.72	1.13	5.12	3.77	8.49	4.47	5.36	43.07
DyRMMA+TMMA	4.65	2.55	8.28	1.15	4.47	4.76	8.86	4.29	5.18	44.09
LBMA+LMMA	4.62	2.62	8.31	1.12	5.23	3.91	12.87	6.11	5.91	50.19
IMAR^2+LMMA	4.54	2.30	8.53	1.14	5.22	3.84	14.13	5.38	6.87	51.45
CIMAR+LMMA	4.24	2.08	8.58	1.21	4.84	2.96	8.55	4.20	4.89	40.35
NIMAR+LMMA	4.60	2.46	8.72	1.20	5.15	4.15	8.56	4.29	5.12	43.39
SMA+LMMA	4.93	2.15	8.77	1.13	5.11	4.18	7.95	4.10	5.29	42.81
DyRMMA+LMMA	4.66	2.65	8.21	1.18	4.52	5.02	17.56	4.49	5.15	52.69
RMMA	8.79	4.60	8.18	1.17	5.72	14.18	15.29	9.79	20.37	87.73
TMMA	4.98	2.54	8.68	1.11	5.08	5.73	9.57	4.55	5.33	47.49
LMMA	4.78	2.53	8.52	1.16	5.03	5.93	9.60	4.11	5.46	46.46

Table 4.13: Energy consumption (in Wh) for Experiment Single in Server ct1. Lower is better.

4.9 Issues with memory pages migrations

Migrating memory pages is an expensive operation given its nature since a lot of synchronisation and copies are required. Furthermore, previous research works have shown that the Linux memory migration mechanism is inefficient [37, 112].

A benchmark has been developed to measure the potential impact of memory migrations, assuming executed in a NUMA system. In this experiment, there are two groups of workers, namely, W^* and W^-, that will perform operations over star and dash operands. Each group of workers will use t threads such that the threads of W^* run in the NUMA node 0, namely ν^*, and W^- run in the opposite node, ν^-. The set of operands, Ω, span through a set of pages, $\Psi = \{\psi_0, \ldots, \psi_{P-1}\}$, allocated in memory such that $\psi_0, \ldots, \psi_{P/2-1}$ are initially placed in ν^*, and pages $\psi_{P/2}, \ldots, \psi_{P-1}$ are placed in ν^-. Operands for each page are shared between workers such that, at the beginning of the experiment, a fraction $s \in [0, 1]$ of the operations over the elements in the pages are local, that is, workers and operands are in the same NUMA node. Operations are executed in random order to avoid the effect of prefetching. After

a given number of operations, o, each page is migrated to the opposite node under a certain probability p. When $s < 0.5$, the migration is expected to be beneficial, when $s \geq 0.5$, the migration should be counterproductive. Note that when $s = 0.5$, migration should also produce bad results because of the overhead of moving pages. Figure 4.31 shows an example of the initial placement of data.

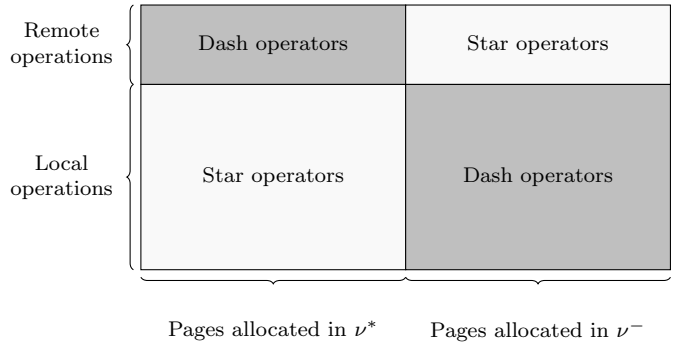


Figure 4.31: Example of the initial placement of data with $s = 0.70$.

Algorithm 4.1 shows the pseudocode of the benchmark. Note that Steps 9 and 16 are parallel loops, also performed at the same time. The source code of the benchmark is available at [113].

Figures 4.32 to 4.37 show the results of this experiment in the Server ctnuma1 for different configurations of the benchmark. Note that figures 4.36 and 4.37 show a row indicating where Thanos would be located in terms of the number of pages migrated.

Figures 4.32 and 4.33—detailed versions are shown in Figures B.1 and B.2—show the four extreme situations and all the intermediate scenarios in between. The upper-left corner shows the scenario where all the memory operations are remote, and pages are not migrated until the end of the benchmark; in the upper-right corner, all memory operations are local and pages are not migrated; in the lower-left corner all pages are remote at the beginning, but pages are migrated at the very first operation; finally, the lower-right corner shows the execution time where all the pages are local, but moved quickly.

The results follow the expected theoretical output, where execution times are negatively affected by remote operations. Thus, the data highlights the importance of performing the migration as soon as possible when the locality is bad, as well as showing the repercussions of bad migration decisions when the locality is good. Furthermore, the delay at which the migrations are performed also has a severe impact, since the sooner a decision is made, the more time it has to affect the outcome. Note that both parameters—locality and delay—have an almost perfect linear influence on execution times.

Algorithm 4.1 Memory migration benchmark.

Input: Threads per node, t.
Total number of pages, P.
Local operands share, s.
Operations to migrate, o.
Probability of migration, p.
Output: Execution time of the benchmark.

1: **procedure** MEMORY MIGRATION BENCHMARK(t, P, s, o, p)
2: $N :=$ Size of each page/Size of each operand ▷ Operands per memory page.
3: $\Omega :=$ set of operands located in P memory pages
4: $\Omega^* := \{\omega_i \in \Omega \mid i \bmod N \leq \frac{sN}{100}, \frac{i}{N} \leq \frac{P}{2}\} \cup \{\omega_i \in \Omega \mid i \bmod N > \frac{sN}{100}, \frac{i}{N} > \frac{P}{2}\}$
5: $\Omega^- := \{\omega_i \in \Omega \mid \omega_i \notin \Omega^*\}$
6: $a^* = 0$
7: $a^- = 0$
8: $\tau_s :=$ time of the start of the benchmark
9: **for each** $\omega_i \in \Omega^*$ **do** ▷ Parallel loop executed by t threads in ν^*.
10: $a^* = a^* + \omega_i$
11: $m = \lfloor i/N \rfloor$
12: **if** Operations over $\psi_m = o$ **then**
13: $\hat{p} =$ random number within $[0, 1]$
14: **if** $\hat{p} \leq p$ **then**
15: Migrate ψ_m to the opposite node
16: **for each** $\omega_i \in \Omega^-$ **do** ▷ Parallel loop executed by t threads in ν^-.
17: $a^- = a^- + \omega_i$
18: $m = \lfloor i/N \rfloor$
19: **if** Operations over $\psi_m = o$ **then**
20: $\hat{p} =$ random number within $[0, 1]$
21: **if** $\hat{p} \leq p$ **then**
22: Migrate ψ_m to the opposite node
23: $\tau_e :=$ time of the end of the benchmark
24: **return** $\tau_e - \tau_s$

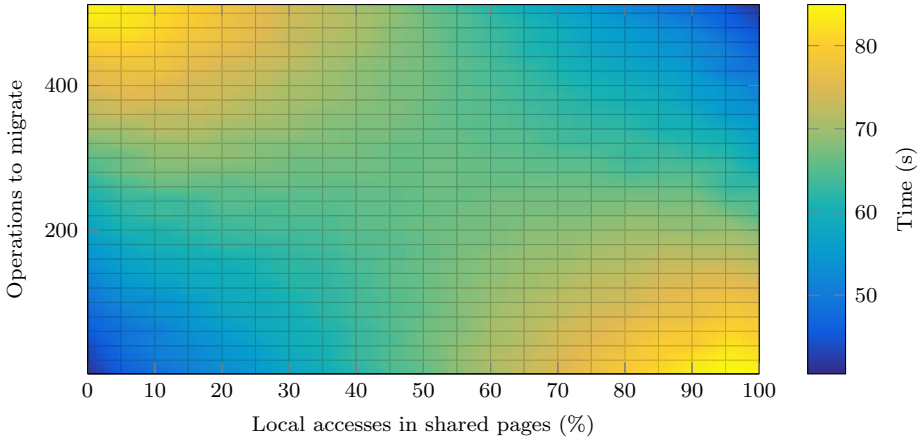


Figure 4.32: Execution time (s) for different amounts of remote pages—over a total of $P = 1\,000\,000$ pages—and operations to migrate a page, with a probability of migration $p = 1$. $t = 1$ thread per worker group. Lower is better.

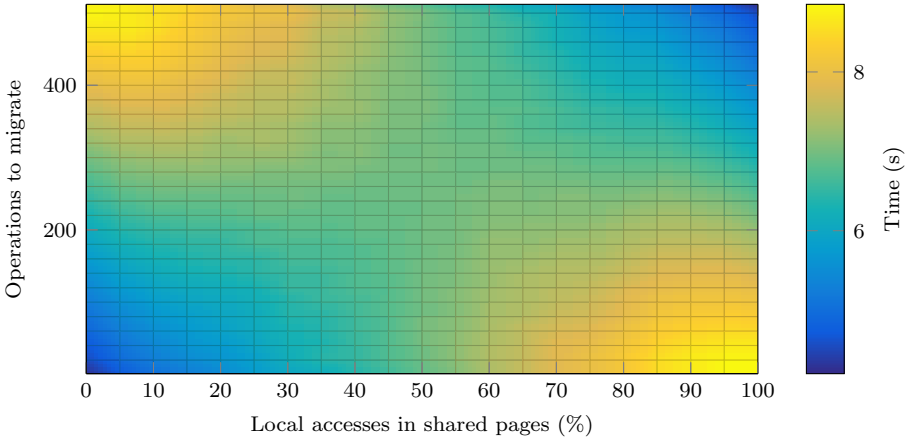


Figure 4.33: Execution time (s) for different amounts of remote pages—over a total of $P = 1\,000\,000$ pages—and operations to migrate a page, with a probability of migration $p = 1$. $t = 10$ threads per worker group. Lower is better.

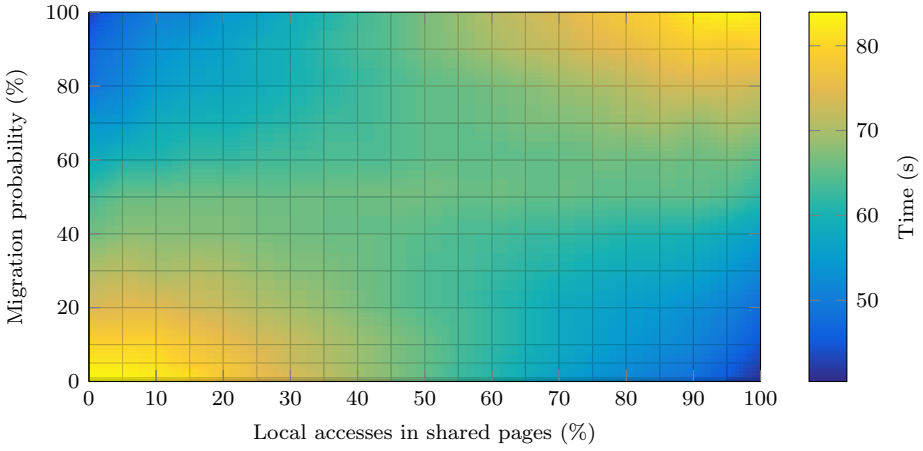


Figure 4.34: Execution time (s) for different amounts of remote pages—over a total of $P = 1\,000\,000$ pages—and chances to migrate a page after $o = 20$ operations. $t = 1$ thread per worker group. Lower is better.

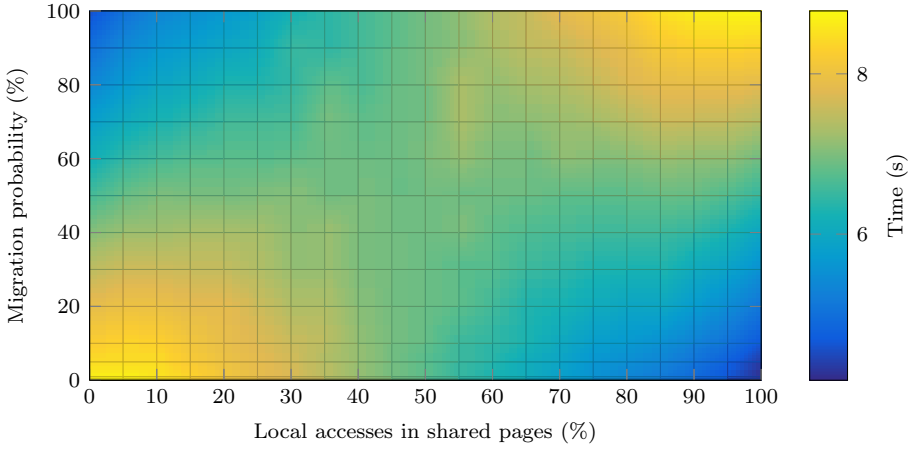


Figure 4.35: Execution time (s) for different amounts of remote pages—over a total of $P = 1\,000\,000$ pages—and chances to migrate a page after $o = 20$ operations. $t = 10$ threads per worker group. Lower is better.

The presence of more threads in each working unit does not affect the behaviour, and execution times just scale almost perfectly with the number of threads involved in the computations.

Figures 4.36 and 4.37 show the results of the benchmark when moving small pages after 20 operations under different probabilities of migration. In the author's opinion, 20 samples is a reasonable amount of information to decide whether a memory page should be migrated or not and its possible destination. Again, the results are coherent with the expected theoretical output, where the higher the number of pages to be migrated, the higher impact of these migrations.

Note that figures 4.36 and 4.37 include a row named Thanos. This row represents the scenario where migrations are handled by Thanos using the samples obtained from the hardware counters. The Thanos row is at the bottom since it is only able to move about 0.05 % of the memory pages due to the scarcity of the information extracted from the hardware counters, which has negligible impact on performance either good or bad. It should be noted that execution times for Thanos are slightly worse than performing no migrations due to the overhead induced by enabling the PMUs in the CPUs.

The main problem of Thanos regarding the migration of memory pages resides in the scarcity of data. Despite that the raw amount of samples regarding memory operations is high enough to have a general overview of how a process is doing in terms of latency or remote accesses, it is not enough to highlight problems related to particular memory pages.

Figures 4.38 to 4.39 show the number of memory pages from which a certain quantity of samples have been gathered during the execution of the benchmark LU.D in Server ctnuma1, which uses 2.40 million pages as reported by the information located at `/proc`.

These figures show the scarcity of the information provided from the hardware performance counters samples. When processing small pages, less than 50 % of the pages have been sampled at least once during the execution of the benchmark. Also, the memory samples received every second cover only 0.30 % of the memory pages present in the system. The situation is aggravated when requiring 10 samples per page since only 0.16 % of the pages have produced such samples at the end of the execution. Every second, the number of pages which received 10 or more samples represents only the 0.01 % of the total.

Treating pages using fTHPs improves the situation slightly. The 93.60 % of the fTHPs have been sampled at least once during the execution. A similar number of fTHPs—93 %—produced 10 or more samples. However, the number of regions sampled per second oscillates significantly during the execution time. Between 10 % and 50 % of the fTHPs are sampled each second, but only 1 % of the pages have been sampled 10 times or more.

Handling 1 % of the pages is usually not enough to produce relevant changes in performance in most scenarios. Note that aiming to have information about every page

Figure 4.36: Execution time (s) for different amounts of remote pages—over a total of $P = 1\,000\,000$ pages—and chances to migrate a page after $o = 20$ operations. $t = 1$ thread per worker group. Lower is better.

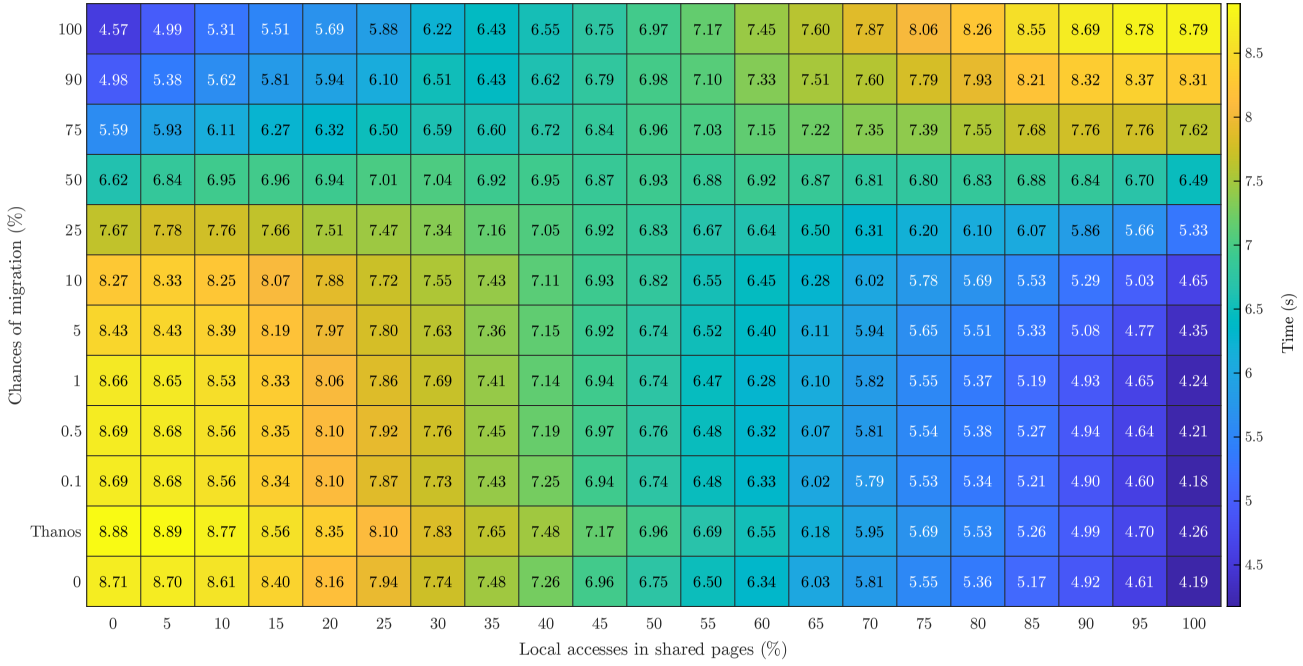


Figure 4.37: Execution time (s) for different amounts of remote pages—over a total of $P = 1\,000\,000$ pages—and chances to migrate a page after $o = 20$ operations. $t = 10$ threads per worker group. Lower is better.

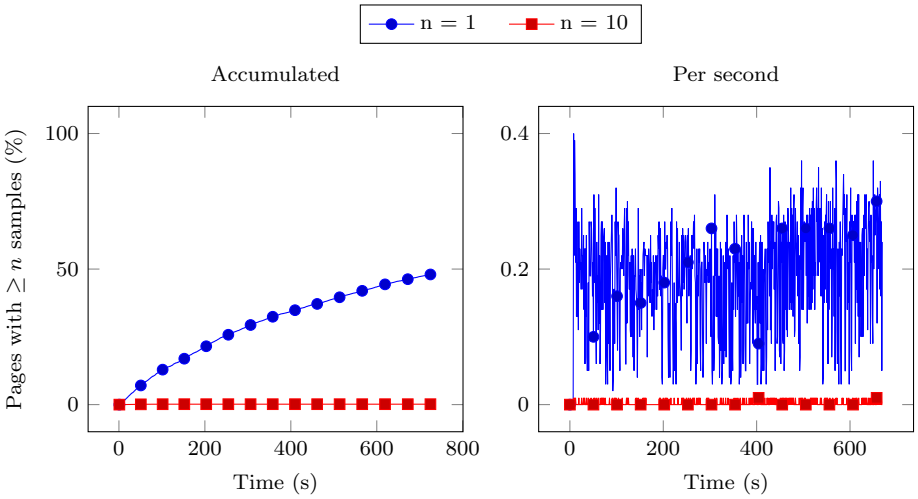


Figure 4.38: Percentage of memory pages with at least n samples for the benchmark LU.D in Server ctnuma1.

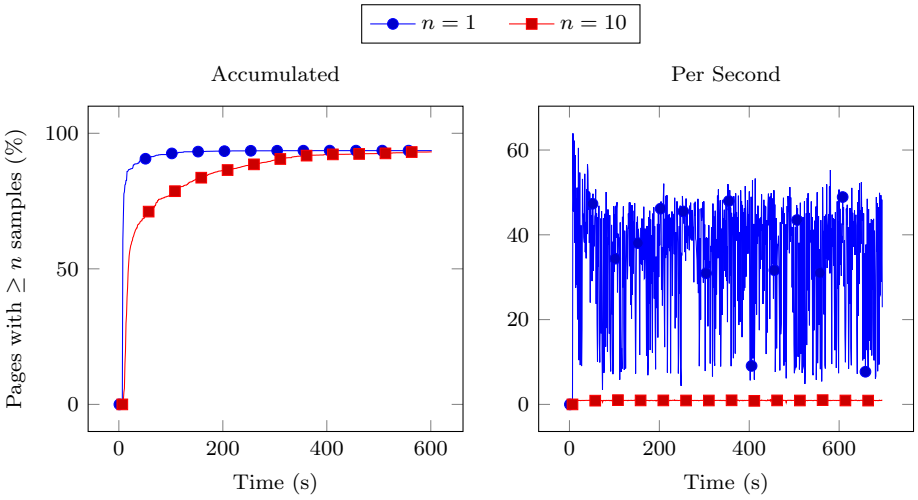


Figure 4.39: Percentage of fTHPs with at least n samples for the benchmark LU.D in Server ctnuma1.

in the system is not realistic either, since most codes exploit locality and only use a reduced set of the memory pages allocated within a short period of time. Furthermore, it is also possible that once a memory page is used, it will not be used again for a long time, or never at all.

Given the numbers, it is possible to state that memory migration is a difficult task for user-space tools. The correct preallocation of resources is of paramount importance in the codes used in the HPC. Considering that samples are only obtained when expensive memory operations are performed, user-space tools can only act *a posteriori*. Therefore, user space tools like Thanos are less likely to improve performance via memory pages migration.

The most significant contributions of this chapter are published and extracted from the subsequent articles:

- O. García Lorenzo, R. Laso Rodríguez, T. Fernández Pena, J. C. Cabaleiro Domínguez, F. Fernández Rivera, and J. Á. Lorenzo del Castillo, "A new hardware counters based thread migration strategy for NUMA systems", in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, E. Deelman, J. Dongarra, and K. Karczewski, Eds., Cham: Springer International Publishing, 2020, pp. 205–216, ISBN: 978-3-030-43222-5. DOI: https://doi.org/10.1007/978-3-030-43222-5_18.
- R. Laso, O. G. Lorenzo, F. F. Rivera, J. C. Cabaleiro, T. F. Pena, and J. A. Lorenzo, "LBMA and IMAR^2: Weighted lottery based migration strategies for NUMA multiprocessing servers", *Concurrency and Computation: Practice and Experience*, vol. 33, no. 11, e5950, 2021. DOI: https://doi.org/10.1002/cpe.5950.
- R. Laso, O. G. Lorenzo, J. C. Cabaleiro, T. F. Pena, J. Á. Lorenzo, and F. F. Rivera, "CIMAR, NIMAR, and LMMA: Novel algorithms for thread and memory migrations in user space on NUMA systems using hardware counters", *Future Generation Computer Systems*, vol. 129, pp. 18–32, 2022, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2021.11.008.

Chapter 5

Conclusions

I have spoken.
— Kuiil, *The Mandalorian.*

This work has faced the challenge of optimising the workload distribution in two different environments, each with its characteristics, requiring different approaches and solutions. This chapter gathers a summary of the problems addressed, the proposals included in this work and the results obtained. Some conclusions and notes on future work are included too.

5.1 CPU and GPU heterogeneous parallelism

The use of GPUs for general-purpose computing boosted the performance in diverse scenarios where parallelism could be exploited. Nevertheless, using only the GPU leaves the CPU idle, which is suboptimal in terms of resource management. For this reason, exploiting heterogeneous parallelism is still a way to be explored in the search for performance.

As shown in Chapter 2, IHP is proposed in this work to maximise the usage of CPU and GPU to speed up the computations, particularly in iterative or time-step methods. The global domain is divided into two subdomains that are assigned to both kinds of computing units. Accordingly to the performance, the subdomains are dynamically resized, so ideally the time spent by the CPU and the GPU is the same. The initial version of the library, IHPv1 assumes a linear workload model, where the execution time grows linearly with the workload. The second version, IHPv2 incorporates different models that are fitted to the real data so the best model is selected. Additionally, IHPv2 includes additional mechanisms to reduce data transfers between CPU and GPU.

Results show that IHPv1 reduces execution times on linear workloads by up to 55 %, compared to the GPU-only implementation. This improvement varies depending

on whether single or double-precision operations are used. It is known that GPUs struggle more with double-precision floating-point operations than CPUs, so this is the scenario where the heterogeneous implementation works better. Compared to other libraries, IHPv1 obtains superior results mainly because of a better calculation of the workload share and reduced data transfers between CPU and GPU.

Furthermore, IHPv2 slightly improve the performance obtained by IHPv1 when non-linear workloads are used. It is interesting to note that IHPv1 still performs well in this scenario since the region in which the optimal workload share is located is locally linear. Thus, once IHPv1 is close enough to the optimal, its performance is similar to IHPv2. It is in the amount of data transfers where IHPv2 improves IHPv1, but it is still a small improvement.

Regarding energy consumption, the heterogeneous solutions are not as efficient as the GPU-only implementations. This is caused by the low efficiency of CPUs compared to GPUs in parallel workloads.

Future work would comprehend the evaluation of these strategies in production code or more complex benchmarks, like finite-element methods. Also, further adjustments could be made in the kind of workloads contemplated in the model fitting of IHPv2 as well as the exploration of automatically providing initial guesses for the workload share. Finally, it would be interesting to explore the use of other accelerators like FPGAs, the use of more than two computing units, or incorporate weights to the workload share, so each piece of work is assigned a different amount of resources.

5.2 NUMA scheduling

NUMA systems present a complex architecture where it is complicated to extract the maximum performance. In particular, it is the relation between execution threads and memory operations where this complexity lies, performance-wise. Locality is key, and local operations should be enforced to avoid increased latency when accessing data from remote nodes. Nevertheless, a balance between local and remote memory operations should be found to avoid potential congestion in the interconnection network.

The current—at the time of writing this work—Linux scheduler, named CFS, is efficient and currently meets the performance expectations in most kinds of systems, something to be praised given the ubiquity of Linux across very different systems. Even though it considers several scheduling domains, distinguishing between physical and logical CPUs, and even NUMA domains, it is still too focused on workload balancing. Interesting patches have been progressively included in the kernel to mitigate that. Particularly, Transparent Huge Pages and NUMA Balancing should be highlighted. The first gathers consecutive memory pages and creates *huge* pages, which allow for more efficient memory handling. The second, periodically unmaps memory pages and migrates them to other NUMA nodes trying to improve locality. Despite the advances

in the CFS, it is believed that there is still a margin for improvement regarding the mapping of threads and memory pages in NUMA system.

In the search for better mappings, this work proposes a collection of algorithms which are explained in Chapter 3. These strategies have been implemented in a user-space tool named Thanos. This tool gathers information about performance, mainly through hardware performance counters (HC), and the different algorithms use that data to decide whether migrations are needed or not, which threads or pages should be migrated and their destination.

Experimental results are presented in Chapter 4. Three experiments have been designed according to three common uses for NUMA systems, namely: Single, Interactive and Queue. In the Experiment Single, only one task is executed with all the resources available. In the Experiment Interactive, several small tasks are executed at arbitrary moments emulating an interactive environment. In the Experiment Queue, the system is kept busy all the time by emulating the behaviour of several users sending tasks to a queue. In the search for completeness, two NUMA servers with different characteristics have been used to carry out the experiments.

First, a comparison of CFS with and without the NB and THP patches have been carried out, showing that enabling both patches can increase performance by up to 47 %, particularly in scenarios where different benchmarks are running simultaneously. Thus, CFS with NB and THP is considered as the Baseline.

Second, a comparison between the Baseline, the common mapping options Interleave and Direct, and the algorithms proposed included in Thanos have been performed. Results show that the Direct mapping is a strong option in multitasking environments, but is far from optimal when running a single benchmark. The performance obtained with the Interleaved mapping differs between systems, where those in which remote latency is smaller can benefit from the increased bandwidth. Otherwise, the performance losses might be too big to be compensated.

Regarding the algorithms proposed in Chapter 3, three particular strategies should be highlighted: CIMAR, NIMAR and SMA. Those algorithms, which migrate threads only, manage to improve the execution times of the Baseline consistently across the three experiments and the two systems. CIMAR and NIMAR are evolutions of the IMAR^2 algorithm and they differ in the way the threads are pinned. On one hand, CIMAR pins threads to particular cores, which enforces a better use of the cache. On the other, NIMAR pins the threads to NUMA nodes, so the workload balance within the node is done by the OS. SMA uses a scoring system to find the global maximum performance, potentially migrating all the threads of the system at each time. Those workloads that make intensive use of the cache see better results with CIMAR, while NIMAR and SMA are more suited to multitasking environments. Performance improvements go up to 46 % in Experiment Single, and up to 11 % in the multitasking experiments.

Little improvements could be achieved by migrating memory pages due to the scarcity of the information provided by the hardware counters. Measurements show

that sampling through HC provides 10 samples or more for only 1 % of the pages. Further experiments have been carried out, noting that migrating that low number of pages is usually negligible when trying to improve performance.

Finally, the impact of the proposed strategies regarding energy has been measured. Data shows a strong correlation between execution times and energy consumption, the latter growing linearly with the execution times. Thus, the algorithms implemented in Thanos produce significant energy savings, up to 40 % for individual tasks and up to 12 % in multitasking environments.

As a matter of future work, new strategies might be developed, mainly combining the decisions of threads with the memory pages of the same process. Also, it should be considered to incorporate the algorithms into the Linux kernel via kernel modules, patches or using other alternatives like ghOSt [114]. The coupling between CFS and the proposals included in this work should improve the decision quality and allow the introduction of other features like the correct preallocation of resources, which is not possible when working in user space.

5.3 Contributions

Attending to the objectives of this PhD dissertation and the aforementioned work and results, it is possible to summarise the main contributions of this PhD dissertation as follows:

- Articles in peer-reviewed journals:
  - R. Laso, O. G. Lorenzo, J. C. Cabaleiro, T. F. Pena, J. Á. Lorenzo, and F. F. Rivera, "CIMAR, NIMAR, and LMMA: Novel algorithms for thread and memory migrations in user space on NUMA systems using hardware counters", *Future Generation Computer Systems*, vol. 129, pp. 18–32, 2022, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2021.11.008.
  - R. Laso, J. C. Cabaleiro, F. F. Rivera, M. C. Muñiz, and J. A. Álvarez-Dios, "IHP: A dynamic heterogeneous parallel scheme for iterative or time-step methods—image denoising as case study", *The Journal of Supercomputing*, vol. 77, no. 1, pp. 95–110, Jan. 2021. DOI: https://doi.org/10.1007/s11227-020-03260-8.
  - R. Laso, O. G. Lorenzo, F. F. Rivera, J. C. Cabaleiro, T. F. Pena, and J. A. Lorenzo, "LBMA and IMAR^2: Weighted lottery based migration strategies for NUMA multiprocessing servers", *Concurrency and Computation: Practice and Experience*, vol. 33, no. 11, e5950, 2021. DOI: https://doi.org/10.1002/cpe.5950.

  - R. Laso, J. C. Cabaleiro, F. F. Rivera, M. C. Muñiz, and J. A. Álvarez-Dios, "Load balanced heterogeneous parallelism for finite difference problems on image denoising", *Computational and Mathematical Methods*, vol. 3, no. 3, e1089, 2021. DOI: https://doi.org/10.1002/cmm4.1089.

- Articles published in international conferences:

  - R. Laso, F. F. Rivera, and J. C. Cabaleiro, "Influence of architectural features of the SNC-4 mode of the Intel Xeon Phi KNL on matrix multiplication", in *Computational Science – ICCS 2019*, J. M. F. Rodrigues, P. J. S. Cardoso, J. Monteiro, R. Lam, V. V. Krzhizhanovskaya, M. H. Lees, J. J. Dongarra, and P. M. Sloot, Eds., Cham: Springer International Publishing, 2019, pp. 483–490, ISBN: 978-3-030-22750-0. DOI: https://doi.org/10.1007/978-3-030-22750-0_41.
  - O. García Lorenzo, R. Laso Rodríguez, T. Fernández Pena, J. C. Cabaleiro Domínguez, F. Fernández Rivera, and J. Á. Lorenzo del Castillo, "A new hardware counters based thread migration strategy for NUMA systems", in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, E. Deelman, J. Dongarra, and K. Karczewski, Eds., Cham: Springer International Publishing, 2020, pp. 205–216, ISBN: 978-3-030-43222-5. DOI: https://doi.org/10.1007/978-3-030-43222-5_18.

- Articles published in national conferences:

  - R. Laso, F. F. Rivera, and J. C. Cabaleiro, "Producto matricial en el Intel Xeon Phi KNL en el modo Sub-NUMA Clustering 4", in *Jornadas SARTECO 2018*, 2018, pp. 129–136.

- Software:

  - R. Laso, J. C. Cabaleiro, F. F. Rivera, M. C. Muñiz, and J. A. Álvarez-Dios. (2019). IHP: Iterative Heterogeneous Parallelism, [Online]. Available: https://gitlab.citius.usc.es/ruben.laso/ihp (visited on 01/19/2023).
  - R. Laso, Ó. G. Lorenzo, F. F. Rivera, J. C. Cabaleiro, T. F. Pena, and J. Á. Lorenzo. (2021). Thanos, [Online]. Available: https://gitlab.citius.usc.es/ruben.laso/migration (visited on 12/29/2022).
  - R. Laso, Ó. G. Lorenzo, F. F. Rivera, J. C. Cabaleiro, T. F. Pena, and J. Á. Lorenzo. (2022). `mmig_bench`, [Online]. Available: https://gitlab.citius.usc.es/ruben.laso/mmigr_bench (visited on 01/19/2023).

Bibliography

[1] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th. USA: Prentice Hall Press, 2014, ISBN: 013359162X.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. Amsterdam: Morgan Kaufmann, 2012, ISBN: 978-0-12-383872-8.

[3] J. Dongarra, M. A. Heroux, and P. Luszczek, "HPCG benchmark: A new metric for ranking high performance computing systems", *Knoxville, Tennessee*, vol. 42, 2015.

[4] J. McCalpin, "Memory bandwidth and machine balance in high performance computers", *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 19–25, Dec. 1995.

[5] ——, (1995). STREAM: Sustainable memory bandwidth in high performance computers, [Online]. Available: https://www.cs.virginia.edu/stream/ (visited on 03/14/2023).

[6] H. Jin, M. Frumkin, and J. Yan, "The OpenMP implementation of NAS parallel benchmarks and its performance", Technical Report NAS-99-011, NASA Ames Research Center, Tech. Rep., 1999.

[7] K. M. Dixit, "The SPEC benchmarks", *Parallel Computing*, vol. 17, no. 10, pp. 1195–1209, 1991, Benchmarking of high performance supercomputers, ISSN: 0167-8191. DOI: https://doi.org/10.1016/S0167-8191(05)80033-X. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016781910580033X.

[8] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures", *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009, ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. [Online]. Available: https://doi.org/10.1145/1498765.1498785.

[9] V. C. Cabezas and M. Püschel, "Extending the roofline model: Bottleneck analysis with microarchitectural constraints", in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014, pp. 222–231. DOI: 10.1109/IISWC.2014.6983061.

[10] N. Ding and S. Williams, "An instruction roofline model for GPUs", in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019, pp. 7–18. DOI: 10.1109/PMBS49563.2019.00007.

[11] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera, "3DyRM: A dynamic roofline model including memory latency information", *The Journal of Supercomputing*, vol. 70, no. 2, pp. 696–708, 2014. [Online]. Available: https://doi.org/10.1007/s11227-014-1163-4.

[12] O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, J. C. Pichel, and F. F. Rivera, "Using an extended roofline model to understand data and thread affinities on NUMA systems", *Annals of Multicore and GPU Programming*, vol. 1, no. 1, pp. 37–48, 2014.

[13] A. Guermouche and A.-C. Orgerie, "Experimental analysis of vectorized instructions impact on energy and power consumption under thermal design power constraints", Télécom Sud Paris, Research Report 2, 2019. [Online]. Available: https://hal.science/hal-02167083.

[14] Arm Limited. (). Arm Big.LITTLE, [Online]. Available: https://www.arm.com/technologies/big-little (visited on 02/09/2023).

[15] A. Mascitti, T. Cucinotta, and M. Marinoni, "An adaptive, utilization-based approach to schedule real-time tasks for arm big.little architectures", *SIGBED Rev.*, vol. 17, no. 1, pp. 18–23, Jul. 2020. DOI: 10.1145/3412821.3412824. [Online]. Available: https://doi.org/10.1145/3412821.3412824.

[16] J. Peddie. (2021). Is it time to rename the GPU?, [Online]. Available: https://www.computer.org/publications/tech-news/chasing-pixels/is-it-time-to-rename-the-gpu (visited on 03/05/2022).

[17] NVIDIA Corporation. (1999). Nvidia launches the world's first graphics processing unit: Geforce 256, [Online]. Available: https://web.archive.org/web/20160412035751/https://www.nvidia.com/object/IO_20020111_5424.html (visited on 03/05/2022).

[18] J. Krüger and R. Westermann, "Linear algebra operators for gpu implementation of numerical algorithms", *ACM Trans. Graph.*, vol. 22, no. 3, pp. 908–916, Jul. 2003, ISSN: 0730-0301. DOI: 10.1145/882262.882363. [Online]. Available: https://doi.org/10.1145/882262.882363.

[19] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: Conjugate gradients and multigrid", *ACM Trans. Graph.*, vol. 22, no. 3, pp. 917–924, Jul. 2003, ISSN: 0730-0301. DOI: 10.1145/882262.882364. [Online]. Available: https://doi.org/10.1145/882262.882364.

[20] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA", *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008, ISSN: 1542-7730. DOI: 10.1145/1365490.1365500. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500.

[21] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems", *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, May 2010, ISSN: 1521-9615. DOI: 10.1109/MCSE.2010.69.

[22] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL", in *2011 International Conference on Parallel Processing*, 2011, pp. 216–225. DOI: 10.1109/ICPP.2011.45.

[23] (2021). SYCL 2020 specification (revision 4), [Online]. Available: https://www.khronos.org/registry/SYCL/specs/sycl-2020/pdf/sycl-2020.pdf (visited on 08/04/2022).

[24] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Jan. 2021, ISBN: 978-1-4842-5573-5. DOI: 10.1007/978-1-4842-5574-2.

[25] Codeplay Software Ltd. (2022). ComputeCpp Community Edition, [Online]. Available: https://developer.codeplay.com/products/computecpp/ce/home/ (visited on 12/28/2022).

[26] GCC Developer Community. (2022). GCC, the GNU Compiler Collection, [Online]. Available: https://gcc.gnu.org/ (visited on 03/30/2022).

[27] NVIDIA Corporation. (Sep. 2020). CUDA Compiler driver NVCC, Reference Guide, [Online]. Available: https://docs.nvidia.com/pdf/CUDA_Compiler_Driver_NVCC.pdf (visited on 04/04/2022).

[28] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation", in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, Mar. 2004, pp. 75–86. DOI: 10.1109/CGO.2004.1281665.

[29] NVIDIA Corporation. (2022). GeForce GTX 3070 Ti, [Online]. Available: https://www.nvidia.com/es-es/geforce/graphics-cards/30-series/rtx-3070-3070ti/ (visited on 05/03/2022).

[30] M. Harris. (Jan. 2013). How to access global memory efficiently in CUDA C/C++ kernels, [Online]. Available: https://devblogs.nvidia.com/how-access-global-memory-efficiently-cuda-c-kernels/ (visited on 12/27/2018).

[31] V. Volkov, "Better performance at lower occupancy", *Proceedings of the GPU Technology Conference, GTC*, vol. 10, Jan. 2015.

[32] NVIDIA Corporation. (2020). NVIDIA HPC SDK, [Online]. Available: https://developer.nvidia.com/hpc-sdk (visited on 05/11/2022).

[33] D. Olsen, G. Lopez, and B. A. Lelbach. (2020). Accelerating standard C++ with GPUs using stdpar, [Online]. Available: https://developer.nvidia.com/blog/accelerating-standard-c-with-gpus-using-stdpar/ (visited on 05/11/2022).

[34] W. J. Bolosky and M. L. Scott, "False sharing and its effect on shared memory performance", in *USENIX Systems on USENIX Experiences with Distributed and Multiprocessor Systems - Volume 4*, ser. Sedms'93, San Diego, California: USENIX Association, 1993, p. 3.

[35] D. Gureya, J. Neto, R. Karimi, J. Barreto, P. Bhatotia, V. Quema, R. Rodrigues, P. Romano, and V. Vlassov, "Bandwidth-aware page placement in NUMA", in *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020, pp. 546–556. DOI: 10.1109/IPDPS47924.2020.00063.

[36] L. Tang, J. Mars, X. Zhang, R. Hagmann, R. Hundt, and E. Tune, "Optimizing google's warehouse scale computers: The NUMA experience", in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 188–197. DOI: 10.1109/HPCA.2013.6522318.

[37] B. Lepers, V. Quema, and A. Fedorova, "Thread and memory placement on NUMA systems: Asymmetry matters", in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, USENIX Assoc., Santa Clara, CA: USENIX Assoc., Jul. 2015, pp. 277–289, ISBN: 978-1-931971-225. [Online]. Available: https://www.usenix.org/conference/atc15/technical-session/presentation/lepers.

[38] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications", in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08, Toronto, Ontario, Canada: Association for Computing Machinery, 2008, pp. 72–81, ISBN: 9781605582825. DOI: 10.1145/1454115.1454128. [Online]. Available: https://doi.org/10.1145/1454115.1454128.

[39] M. Dashti, A. Fedorova, J. Funston, F. Gaud, R. Lachaize, B. Lepers, V. Quema, and M. Roth, "Traffic management: A holistic approach to memory placement on NUMA systems", *SIGPLAN Not.*, vol. 48, no. 4, pp. 381–394, Mar. 2013, ISSN: 0362-1340. DOI: 10.1145/2499368.2451157. [Online]. Available: https://doi.org/10.1145/2499368.2451157.

[40] R. Laso, F. F. Rivera, and J. C. Cabaleiro, "Influence of architectural features of the SNC-4 mode of the Intel Xeon Phi KNL on matrix multiplication", in *Computational Science – ICCS 2019*, J. M. F. Rodrigues, P. J. S. Cardoso, J. Monteiro, R. Lam, V. V. Krzhizhanovskaya, M. H. Lees, J. J. Dongarra, and P. M. Sloot, Eds., Cham: Springer International Publishing, 2019, pp. 483–490, ISBN: 978-3-030-22750-0. DOI: https://doi.org/10.1007/978-3-030-22750-0_41.

[41] ——, "Producto matricial en el Intel Xeon Phi KNL en el modo Sub-NUMA Clustering 4", in *Jornadas SARTECO 2018*, 2018, pp. 129–136.

[42] Intel Corporation. (2016). Intel Xeon Processor E5-4620 v4 processor, [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/93800/intel-xeon-processor-e54620-v4-25m-cache-2-10-ghz/specifications.html (visited on 01/31/2023).

[43] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition 2nd Edition*, 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2016, ISBN: 0128091940.

[44] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu, "Knights Landing: Second-generation Intel Xeon Phi product", *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar. 2016, ISSN: 0272-1732. DOI: 10.1109/MM.2016.25.

[45] A. B. Yoo, M. A. Jette, and M. Grondona, "SLURM: Simple Linux utility for resource management", in *Job Scheduling Strategies for Parallel Processing*, D. Feitelson, L. Rudolph, and U. Schwiegelshohn, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 44–60, ISBN: 978-3-540-39727-4.

[46] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "OmpSs: A proposal for programming heterogeneous multi-core architectures", *Parallel Processing Letters*, vol. 21, no. 02, pp. 173–193, 2011. DOI: 10.1142/S0129626411000151. [Online]. Available: https://doi.org/10.1142/S0129626411000151.

[47] D. Lipari, "The SLURM scheduler design", in *SLURM User Group Meeting, Oct*, vol. 9, 2012, p. 52.

[48] D. Bovet and M. Cesati, *Understanding The Linux Kernel*. Oreilly; Associates Inc, 2005, ISBN: 0596005652.

[49] M. Garland, S. Le Grand, J. Nickolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with CUDA", *IEEE Micro*, vol. 28, no. 4, pp. 13–27, 2008. DOI: 10.1109/MM.2008.57.

[50] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu, "Program optimization space pruning for a multithreaded GPU", in *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '08, Boston, MA, USA: Association for Computing Machinery, 2008, pp. 195–204, ISBN: 9781595939784. DOI: 10.1145/1356058.1356084. [Online]. Available: https://doi.org/10.1145/1356058.1356084.

[51] D. Komatitsch, D. Michéa, and G. Erlebacher, "Porting a high-order finite-element earthquake modeling application to NVIDIA graphics cards using CUDA", *Journal of Parallel and Distributed Computing*, vol. 69, no. 5, pp. 451–460, 2009, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2009.01.006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731509000069.

[52] G. Markall, A. Slemmer, D. Ham, P. Kelly, C. Cantwell, and S. Sherwin, "Finite element assembly strategies on multi-core and many-core architectures", *International Journal for Numerical Methods in Fluids*, vol. 71, no. 1, pp. 80–97, 2013.

[53] R. Raina, A. Madhavan, and A. Y. Ng, "Large-scale deep unsupervised learning using graphics processors", in *Proceedings of the 26th Annual International Conference on Machine Learning*, ser. ICML '09, Montreal, Quebec, Canada: Association for Computing Machinery, 2009, pp. 873–880, ISBN: 9781605585161. DOI: 10.1145/1553374.1553486. [Online]. Available: https://doi.org/10.1145/1553374.1553486.

[54] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Y. Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org, [Online]. Available: https://www.tensorflow.org/ (visited on 12/15/2022).

[55] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An imperative style, high-performance deep learning library", in *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., vol. 32, Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[56] M. Papadrakakis, G. Stavroulakis, and A. Karatarakis, "A new era in scientific computing: Domain decomposition methods in hybrid CPU–GPU architectures", *Computer Methods in Applied Mechanics and Engineering*, vol. 200, no. 13, pp. 1490–1508, 2011, ISSN: 0045-7825. DOI: https://doi.org/10.1016/j.cma.2011.01.013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045782511000235.

[57] B. Pérez, J. L. Bosque, and R. Beivide, "Simplifying programming and load balancing of data parallel applications on heterogeneous systems", in *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, ser. GPGPU '16, Barcelona, Spain: ACM, 2016, pp. 42–51, ISBN: 978-1-4503-4195-0. DOI: 10.1145/2884045.2884051. [Online]. Available: http://doi.acm.org/10.1145/2884045.2884051.

[58] R. Kaleem, R. Barik, T. Shpeisman, C. Hu, B. T. Lewis, and K. Pingali, "Adaptive heterogeneous scheduling for integrated GPUs", in *2014 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Aug. 2014, pp. 151–162. DOI: 10.1145/2628071.2628088.

[59] R. Barik, R. Kaleem, D. Majeti, B. T. Lewis, T. Shpeisman, C. Hu, Y. Ni, and A.-R. Adl-Tabatabai, "Efficient mapping of irregular C++ applications to integrated GPUs", in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14, Orlando, FL, USA: Association for Computing Machinery, 2014, pp. 33–43, ISBN: 9781450326704. DOI: 10.1145/2581122.2544165. [Online]. Available: https://doi.org/10.1145/2581122.2544165.

[60] A. Navarro, F. Corbera, A. Rodriguez, A. Vilches, and R. Asenjo, "Heterogeneous `parallel_for` template for CPU–GPU chips", *International Journal of Parallel Programming*, vol. 47, no. 2, pp. 213–233, Apr. 2019, ISSN: 1573-7640. DOI: 10.1007/s10766-018-0555-0. [Online]. Available: https://doi.org/10.1007/s10766-018-0555-0.

[61] A. Vilches, R. Asenjo, A. Navarro, F. Corbera, R. Gran, and M. Garzarán, "Adaptive partitioning for irregular applications on heterogeneous CPU-GPU chips", *Procedia Computer Science*, vol. 51, pp. 140–149, 2015, International Conference On Computational Science, ICCS 2015, ISSN: 1877-0509. DOI: http

s://doi.org/10.1016/j.procs.2015.05.213. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1877050915010212.

[62] C. Pheatt, "Intel Threading Building Blocks", *J. Comput. Sci. Coll.*, vol. 23, no. 4, pp. 298–298, Apr. 2008, ISSN: 1937-4771. [Online]. Available: http://dl.acm.org/citation.cfm?id=1352079.1352134.

[63] J. C. Romero, A. Navarro, A. Rodríguez, and R. Asenjo, "SkyFlow: Heterogeneous streaming for skyline computation using FlowGraph and SYCL", *Future Generation Computer Systems*, vol. 141, pp. 269–283, 2023, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2022.11.021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X2200382X.

[64] J. C. Romero, A. Navarro, A. Vilches, A. Rodríguez, F. Corbera, and R. Asenjo, "Efficient heterogeneous matrix profile on a CPU + high performance FPGA with integrated HBM", *Future Generation Computer Systems*, vol. 125, pp. 10–23, 2021, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2021.06.025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167739X2100220X.

[65] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming", *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, Jan. 1998, ISSN: 1070-9924. DOI: 10.1109/99.660313. [Online]. Available: https://doi.org/10.1109/99.660313.

[66] R. Laso, J. C. Cabaleiro, F. F. Rivera, M. C. Muñiz, and J. A. Álvarez-Dios. (2019). IHP: Iterative Heterogeneous Parallelism, [Online]. Available: https://gitlab.citius.usc.es/ruben.laso/ihp (visited on 01/19/2023).

[67] R. Laso, J. C. Cabaleiro, F. F. Rivera, M. C. Muñiz, and J. A. Álvarez-Dios, "IHP: A dynamic heterogeneous parallel scheme for iterative or time-step methods—image denoising as case study", *The Journal of Supercomputing*, vol. 77, no. 1, pp. 95–110, Jan. 2021. DOI: https://doi.org/10.1007/s11227-020-03260-8.

[68] R. Laso, J. C. Cabaleiro, F. F. Rivera, M. C. Muñiz, and J. A. Álvarez-Dios, "Load balanced heterogeneous parallelism for finite difference problems on image denoising", *Computational and Mathematical Methods*, vol. 3, no. 3, e1089, 2021. DOI: https://doi.org/10.1002/cmm4.1089.

[69] B. Tang, G. Sapiro, and V. Caselles, "Color image enhancement via chromaticity diffusion", *IEEE Transactions on Image Processing*, vol. 10, no. 5, pp. 701–707, May 2001, ISSN: 1057-7149. DOI: 10.1109/83.918563.

[70] Intel Corporation. (2017). Intel core i5-7600 processor, [Online]. Available: https://ark.intel.com/products/97150/Intel-Core-i5-7600-Processor-6M-Cache-up-to-4_10-GHz (visited on 06/07/2021).

[71] NVIDIA Corporation. (2018). GeForce GTX 1050 Ti, [Online]. Available: https://www.geforce.com/hardware/desktop-gpus/geforce-gtx-1050-ti/specifications (visited on 07/18/2022).

[72] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010, ISBN: 0131387685.

[73] Intel Corporation. (2015). Intel Core i7-75700HQ processor, [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/87716/intel-core-i75700hq-processor-6m-cache-up-to-3-50-ghz.html (visited on 07/18/2022).

[74] I. Molnar. (2007). CFS scheduler, [Online]. Available: https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt (visited on 09/22/2021).

[75] M. Kalin. (Feb. 2019). CFS: Completely fair process scheduling in Linux, [Online]. Available: https://opensource.com/article/19/2/fair-scheduling-linux (visited on 09/23/2021).

[76] J. Corbet. (Jul. 2019). Scheduling domains, [Online]. Available: https://www.kernel.org/doc/html/v5.8/_sources/scheduler/sched-domains.rst.txt (visited on 09/27/2021).

[77] J.-P. Lozi, B. Lepers, J. Funston, F. Gaud, V. Quéma, and A. Fedorova, "The Linux scheduler: A decade of wasted cores", in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16, London, United Kingdom: Association for Computing Machinery, 2016, ISBN: 9781450342407. DOI: 10.1145/2901318.2901326. [Online]. Available: https://doi.org/10.1145/2901318.2901326.

[78] J. Corbet. (Apr. 2004). Scheduling domains, [Online]. Available: https://opensource.com/article/19/2/fair-scheduling-linux (visited on 09/27/2021).

[79] ——, (Mar. 2012). AutoNUMA: The other approach to NUMA scheduling, [Online]. Available: https://lwn.net/Articles/488709/ (visited on 09/27/2021).

[80] R. van Riel and V. Chegu, "Automatic NUMA balancing", Red Hat Summit, 2014, [Online]. Available: https://www.linux-kvm.org/images/7/75/01x07b-NumaAutobalancing.pdf.

[81] M. Gorman. (2010). Huge pages part 1 (introduction), [Online]. Available: https://lwn.net/Articles/374424/ (visited on 05/25/2022).

[82] ——, (2010). Huge pages part 4: Benchmarking with huge pages, [Online]. Available: https://lwn.net/Articles/378641/ (visited on 05/25/2022).

[83] J. Corbet. (Jan. 19, 2011). Transparent huge pages in 2.6.38, [Online]. Available: https://lwn.net/Articles/423584/ (visited on 08/19/2022).

[84] M. Gorman. (2010). Re: Transparent hugepage support #33, [Online]. Available: https://lwn.net/Articles/423590/ (visited on 08/19/2022).

[85] A. Kleen, C. Wickman, C. Lameter, and L. Schermerhorn. (Jul. 23, 2014). Numactl, [Online]. Available: https://github.com/numactl/numactl (visited on 12/29/2022).

[86] C. Lameter, "Local and remote memory: Memory in a Linux/NUMA system", in *Linux Symposium*, 2006, pp. 1–25.

[87] M. Diener, E. H. Cruz, P. O. Navaux, A. Busse, and H.-U. Heiß, "KMAF: Automatic kernel-level management of thread and data affinity", in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14, Edmonton, AB, Canada: Association for Computing Machinery, 2014, pp. 277–288, ISBN: 9781450328098. DOI: 10.1145/2628071.2628085. [Online]. Available: https://doi.org/10.1145/2628071.2628085.

[88] I. Di Gennaro, A. Pellegrini, and F. Quaglia, "OS-based NUMA optimization: Tackling the case of truly multi-thread applications with non-partitioned virtual page accesses", in *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 291–300. DOI: 10.1109/CCGrid.2016.91.

[89] M.-L. Chiang, C.-J. Yang, and S.-W. Tu, "Kernel mechanisms with dynamic task-aware scheduling to reduce resource contention in NUMA multi-core systems", *Journal of Systems and Software*, vol. 121, pp. 72–87, 2016, ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2016.08.038. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121216301376.

[90] M.-L. Chiang, S.-W. Tu, W.-L. Su, and C.-W. Lin, "Enhancing inter-node process migration for load balancing on Linux-based NUMA multicore systems", in *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, vol. 2, 2018, pp. 394–399.

[91] M.-L. Chiang, W.-L. Su, S.-W. Tu, and Z.-W. Lin, "Memory-aware kernel mechanism and policies for improving internode load balancing on NUMA systems", *Software: Practice and Experience*, vol. 49, no. 10, pp. 1485–1508, 2019.

[92] M. Agung, M. A. Amrizal, R. Egawa, and H. Takizawa, "DeLoc: A locality and memory-congestion-aware task mapping method for modern NUMA systems", *IEEE Access*, vol. 8, pp. 6937–6953, 2020. DOI: 10.1109/ACCESS.2019.2963726.

[93] R. Laso, Ó. G. Lorenzo, F. F. Rivera, J. C. Cabaleiro, T. F. Pena, and J. Á. Lorenzo. (2021). Thanos, [Online]. Available: https://gitlab.citius.usc.es/ruben.laso/migration (visited on 12/29/2022).

[94] Intel Corporation. (2017). Intel 64 and IA-32 Architectures Software Developer Manuals, [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (visited on 02/10/2023).

[95] S. Eranian, *Perfmon2: A standard performance monitoring interface for Linux*, http://perfmon2.sf.net/perfmon2-20080124.pdf, HP Labs, HP Labs, 2008.

[96] S. Huband, P. Hingston, L. Barone, and L. While, "A review of multiobjective test problems and a scalable test problem toolkit", *IEEE Transactions on Evolutionary Computation*, vol. 10(5), pp. 477–506, 2006. [Online]. Available: https://doi.org/10.1109/TEVC.2005.861417.

[97] M. Braun, L. Heling, P. Shukla, and H. Schmeck, "Multimodal scalarized preferences in multi-objective optimization", in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '17, GECCO, Berlin, Germany: Association for Computing Machinery, 2017, pp. 545–552, ISBN: 9781450349208. DOI: 10.1145/3071178.3079189. [Online]. Available: https://doi.org/10.1145/3071178.3079189.

[98] C.-L. Hwang, A. Syed, and M. Masud, *Multiple objective decision making, methods and applications: a state-of-the-art survey*. Springer-Verlag, 2012, ISBN: 978-0-387-09111-2.

[99] R. Laso, O. G. Lorenzo, F. F. Rivera, J. C. Cabaleiro, T. F. Pena, and J. A. Lorenzo, "LBMA and IMAR^2: Weighted lottery based migration strategies for NUMA multiprocessing servers", *Concurrency and Computation: Practice and Experience*, vol. 33, no. 11, e5950, 2021. DOI: https://doi.org/10.1002/cpe.5950.

[100] R. Laso, O. G. Lorenzo, J. C. Cabaleiro, T. F. Pena, J. Á. Lorenzo, and F. F. Rivera, "CIMAR, NIMAR, and LMMA: Novel algorithms for thread and memory migrations in user space on NUMA systems using hardware counters", *Future Generation Computer Systems*, vol. 129, pp. 18–32, 2022, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2021.11.008.

[101] P. J. M. van Laarhoven and E. H. L. Aarts, "Simulated annealing", in *Simulated Annealing: Theory and Applications*. Dordrecht: Springer Netherlands, 1987, pp. 7–15, ISBN: 978-94-015-7744-1. DOI: 10.1007/978-94-015-7744-1_2. [Online]. Available: https://doi.org/10.1007/978-94-015-7744-1_2.

[102] O. García Lorenzo, R. Laso Rodríguez, T. Fernández Pena, J. C. Cabaleiro Domínguez, F. Fernández Rivera, and J. Á. Lorenzo del Castillo, "A new hardware counters based thread migration strategy for NUMA systems", in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, E. Deelman, J. Dongarra, and K. Karczewski, Eds., Cham: Springer International Publishing, 2020, pp. 205–216, ISBN: 978-3-030-43222-5. DOI: https://doi.org/10.1007/978-3-030-43222-5_18.

[103] V. Viswanathan, K. Kumar, T. Willhalm, P. Lu, B. Filipiak, and S. Sakthivelu. (2022). Intel Memory Latency Checker v3.9a, [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html (visited on 04/04/2022).

[104] Intel Corporation. (2019). Intel Xeon Gold 6248 processor, [Online]. Available: https://www.intel.com/content/www/us/en/products/sku/192446/intel-xeon-gold-6248-processor-27-5m-cache-2-50-ghz/specifications.html (visited on 01/31/2023).

[105] M. A. Frumkin and L. Shabano, "Arithmetic data cube as a data intensive benchmark", 2003. [Online]. Available: https://ntrs.nasa.gov/citations/20030022715 (visited on 02/10/2023).

[106] D. Bailey, E. Barszcz, L. Dagum, and H. Simon, "NAS parallel benchmark results", *IEEE Parallel & Distributed Technology: Systems & Applications*, vol. 1, no. 1, pp. 43–51, 1993. DOI: 10.1109/88.219861.

[107] H. Feng, R. F. Van der Wijngaart, R. Biswas, and C. Mavriplis, "Unstructured adaptive (UA) NAS parallel benchmark, version 1.0", *NASA Technical Report NAS-04*, vol. 6, 2004.

[108] (2021). Intel Advisor, [Online]. Available: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/advisor.html (visited on 06/07/2021).

[109] A. C. De Melo, "The new Linux "perf" tools", in *Slides from Linux Kongress*, vol. 18, 2010, pp. 1–42.

[110] C. Lameter, "NUMA (non-uniform memory access): An overview", *ACM Queue*, vol. 11, no. 7, p. 40, 2013. [Online]. Available: https://queue.acm.org/detail.cfm?id=2513149.

[111] A. Rane and D. Stanzione, "Experiences in tuning performance of hybrid MPI/OpenMP applications on quad-core systems", in *Proc. of 10th LCI Int'l Conference on High-Performance Clustered Computing*, 2009, pp. 1–10.

[112] J. Funston, M. Lorrillere, A. Fedorova, B. Lepers, D. Vengerov, J.-P. Lozi, and V. Quéma, "Placement of virtual containers on NUMA systems: A practical and comprehensive model", in *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '18, Boston, MA, USA: USENIX Association, 2018, pp. 281–293, ISBN: 9781931971447.

[113] R. Laso, Ó. G. Lorenzo, F. F. Rivera, J. C. Cabaleiro, T. F. Pena, and J. Á. Lorenzo. (2022). `mmig_bench`, [Online]. Available: https://gitlab.citius.usc.es/ruben.laso/mmigr_bench (visited on 01/19/2023).

[114] J. T. Humphries, N. Natu, A. Chaugule, O. Weisse, B. Rhoden, J. Don, L. Rizzo, O. Rombakh, P. Turner, and C. Kozyrakis, "GhOSt: Fast & flexible user-space delegation of linux scheduling", ser. SOSP '21, Virtual Event, Germany: Association for Computing Machinery, 2021, pp. 588–604, ISBN: 9781450387095. DOI: 10.1145/3477132.3483542. [Online]. Available: https://doi.org/10.1145/3477132.3483542.

List of Figures

List of Tables

List of Listings

List of Algorithms

Acronyms

3DyRM 3Dynamic RM 4, 16, 67, 68, 73

API Application Programming Interface 23

BT Block Tri-diagonal 16, 17, 93, 94, 96, 98–101, 104, 106, 107, 110, 112, 117, 123, 154, 177–182

BWAP Bandwidth-Aware Page Placement 64, 65

cc-NUMA cache-coherent NUMA 25, *see* NUMA

CESGA Centro de Supercomputación de Galicia 5, 95

CFD computational fluid dynamics 93, 94

CFS Completely Fair Scheduler 3–6, 10, 59, 60, 91, 97–99, 134–136

CG Conjugate Gradient 17, 93, 94, 96, 98–102, 104, 107, 108, 110–113, 117, 119, 123, 154, 155, 177–182

CIMAR Core-aware Interchange and Migration Algorithm with performance Record 4, 6, 75, 77–79, 100–102, 104, 105, 107, 108, 110–112, 114, 116–118, 121, 123, 135, 154, 155, 163, 177–182, *see* IMAR

CPU Central Processing Unit 1–3, 5, 9–12, 14–16, 18–27, 29–33, 35, 37–39, 43, 46–54, 59–63, 67, 77, 93, 104, 106, 119, 121, 128, 133, 134, 153, 154, 158, 159, 171–175

CRA Completely Random Algorithm 4, 72, 85, 100, 101, 104, 107, 110, 112, 116, 117, 123, 163, 177–182

CUDA Compute Unified Device Architecture 1, 19–23, 29, 31, 44, 46, 153, 158, 161, 171–174

DC Arithmetic Data Cube 17, 93, 94

DDR Double Data Rate 44, 45

DeLoc decongested locality 64, 65

DRAM Dynamic Random Access Memory 15, 16, 25, 65, 66, 68, 91, 92, *see* RAM

DyRMMA 3DyRM Migration Algorithm 4, 83, 85, 100, 101, 104, 107, 110, 112, 114, 117, 123, 155, 163, 177–182, *see* 3DyRM

EP Embarrasingly Parallel 16, 17, 93–95, 98–101, 104, 112, 117, 123, 177, 178, 181, 182

EWMA Exponentially Weighted Moving Average 36

FEM finite element method 30, 70

FLOPS floating-point operations per second 15, 16

FPGA Field Programmable Gate Array 18, 19, 23, 31, 134

FT 3-D Fast Fourier Transform 16, 17, 94, 98–101, 104, 112, 117, 123, 177, 178, 181, 182

fTHP fake Transparent Huge Page 70, 71, 128, 131, 156, *see* THP

GCC GNU Compiler Collection 20, 44, 45

GPGPU general-purpose computing on GPUs 1, 9, 12, 19, 22, 23, 29, 30, *see* GPU

GPU Graphics Processing Unit 1–3, 9–16, 18–23, 27, 29–33, 35, 37–39, 43, 45–51, 53, 54, 133, 134, 153, 158, 159, 171–175

HBM High Bandwidth Memory 16

HC hardware performance counters 4, 7, 10, 67, 99, 100, 128, 135, 136, 157

HPC high-performance computing 5, 9–11, 13–15, 18, 23, 31, 121, 132

HPCG High Performance Conjugate Gradient 15

I/O Input/Output 93, 94, 119

IHP Iterative Heterogeneous Parallelism 1, 2, 10, 12, 27, 31–36, 38, 40, 43–46, 48–51, 54, 133, 153

IHPv1 IHP version 1 2, 3, 35, 45, 47–51, 53–56, 133, 134, 157–159, 171–175, *see* IHP

IHPv2 IHP version 2 2, 3, 35, 45, 47–51, 133, 134, 158, 159, 171–175, *see* IHP

IMAR² Interchange and Migration Algorithm with performance Record and Rollback 4, 6, 73, 75–77, 79, 100–102, 104, 107, 108, 110, 112, 114, 116–118, 123, 135, 154, 155, 163, 177–182

IS Integer Sort 17, 94, 95, 98–101, 104, 112, 117, 118, 123, 155, 177, 178, 181, 182

kMAF kernel Memory Affinity Framework 64, 65

LBMA Lottery-Based Migration Algorithm 4, 72–75, 100, 101, 104, 107, 110, 112, 114, 117, 123, 155, 163, 177–182

LLC last level cache 23, 94, 95, 154

LMMA Latency Memory Migration Algorithm 5, 87–89, 101, 103, 104, 106, 107, 110, 112, 117, 119, 123, 154, 155, 163, 177–182

LU Lower-Upper Gauss-Seidel 17, 94, 96, 98–102, 104, 105, 107–110, 112, 114, 117, 121, 123, 128, 131, 154–156, 177–182

LWMA Linearly Weighted Moving Average 36

MG Multi-Grid 17, 94, 98–101, 104, 112, 117, 123, 177, 178, 181, 182

MSE Mean Squared Error 39

MVAS Multi-View Address Space 64, 65

NB NUMA Balancing 3–5, 10, 61, 97–99, 134, 135, *see* NUMA

nc-NUMA non cache-coherent NUMA *see* NUMA

NIMAR Node-aware Interchange and Migration Algorithm with performance Record 4, 6, 77, 79, 80, 101, 102, 104, 105, 107, 108, 110, 112, 114–117, 121, 123, 135, 154, 155, 163, 177–182, *see* IMAR

NPB NASA Advanced Supercomputing Parallel Benchmarks 5, 15–17, 26, 92, 94, 95, 97, 99, 100, 153, 154, *see* NAS

NUMA Non-Uniform Memory Access 1, 3–6, 9–14, 16, 17, 23–27, 59–62, 64, 65, 67, 69–71, 73, 77, 79, 81, 86, 87, 89, 91, 95, 97–100, 123, 134, 135, 153, 154

NVCC NVIDIA CUDA Compiler 20, 44, *see* CUDA

OI operational intensity 15, 17

OoO out-of-order 16

OS operating system 10, 18, 26, 62–64, 77, 121, 135

PARSEC Princeton Application Repository for Shared-Memory Computers 64

PDE partial differential equation 40, 42

PEBS Precise Event Based Sampling 67, 68

PMU performance monitoring unit 63, 128

RM Roofline Model 15, 16

RMMA Random Memory Migration Algorithm 5, 85, 86, 101, 103, 104, 107, 110, 112, 116, 117, 123, 154, 163, 177–182

SA simulated annealing 81, 83

SDK Software Development Kit 23

SIMD single instruction, multiple data 15, 20

SIMT single instruction, multiple thread 20, 22

SM streaming multi-processors 20, 22

SMA Score Maximisation Algorithm 4, 6, 81–84, 101, 102, 104, 107–110, 112, 114–117, 121, 123, 135, 154, 155, 163, 177–182

SP Scalar Penta-diagonal 17, 94, 96, 98–101, 103–105, 107, 108, 110, 112, 114–118, 123, 154, 155, 177–182

SPEC Standard Performance Evaluation Corporation 15

TBB Intel Thread Building Blocks 30, 31, 44, 45, 51, 158, 159, 174, 175

Thanos Thread & memory migration Algorithms for NUMA Optimised Scheduling 4, 5, 7, 10, 12, 27, 65, 67, 70–72, 91, 92, 98, 104, 106, 108, 111, 113, 116, 124, 128, 132, 135, 136

THP Transparent Huge Page 3, 5, 10, 62, 70, 71, 97–99, 134, 135

TLB Translation Lookaside Buffer 3, 62

TMMA Threshold Memory Migration Algorithm 5, 86–88, 101, 103, 104, 107, 110, 112, 117, 119, 123, 154, 155, 163, 177–182

UA Unstructured Adaptive 17, 94, 98–101, 104, 112, 117, 123, 177, 178, 181, 182

UMA Uniform Memory Access 9, 14, 23–25, 153

WMSE Weighted Mean Squared Error 39

Appendix A

Results of IHP

		Brightness		Chromaticity	
		Isotropic	Anisotropic	Isotropic	Anisotropic
OpenMP		**23.12**	**333.64**	**318.96**	**1010.56**
CUDA		**2.59**	**4.64**	**32.16**	**84.95**
IHPv1	CPU	2.59	5.28	31.06	83.39
	GPU	2.60	4.85	31.14	80.92
	Copy	0.06	0.37	0.76	3.94
	Total	**2.59**	**5.28**	**31.06**	**83.39**
IHPv2	CPU	2.26	0.42	30.02	80.39
	GPU	2.42	4.54	30.45	80.36
	Copy	0.07	0.02	0.65	0.07
	Total	**2.60**	**4.85**	**31.14**	**80.92**

Table A.1: Execution time (in seconds) of CPU-only (with OpenMP), GPU-only (with CUDA), IHPv1 (OpenMP + CUDA), and IHPv2 (OpenMP + CUDA) implementations using linear workloads on Desktop system. Single-precision computations. Total execution times noted in **bold**.

		Brightness		Chromaticity	
		Isotropic	Anisotropic	Isotropic	Anisotropic
OpenMP		**12.59**	**168.09**	**161.43**	**509.77**
CUDA		**1.69**	**2.98**	**17.60**	**43.75**
IHPv1	CPU	1.90	3.53	17.85	44.45
	GPU	1.92	3.19	17.79	43.76
	Copy	0.10	0.15	0.85	1.33
	Total	**1.90**	**3.53**	**17.85**	**44.45**
IHPv2	CPU	1.49	0.37	16.74	42.34
	GPU	1.65	2.88	16.95	42.56
	Copy	0.16	0.03	0.64	1.74
	Total	**1.92**	**3.19**	**17.79**	**43.76**

Table A.2: Execution time (in seconds) of CPU-only (with OpenMP), GPU-only (with CUDA), IHPv1 (OpenMP + CUDA), and IHPv2 (OpenMP + CUDA) implementations using logarithmic workloads on Desktop system. Single-precision computations. Total execution times noted in **bold**.

		Brightness		Chromaticity	
		Isotropic	Anisotropic	Isotropic	Anisotropic
OpenMP		**108.01**	**1661.45**	**1587.75**	**5046.31**
CUDA		**1.75**	**21.42**	**159.09**	**421.41**
IHPv1	CPU	1.98	20.67	121.44	325.93
	GPU	1.94	20.39	119.91	321.50
	Copy	0.13	0.41	2.13	4.45
	Total	**1.98**	**20.67**	**121.44**	**325.93**
IHPv2	CPU	1.61	19.98	118.25	319.35
	GPU	1.71	19.91	118.47	319.60
	Copy	0.11	0.02	0.09	0.08
	Total	**1.94**	**20.39**	**119.91**	**321.50**

Table A.3: Execution time (in seconds) of CPU-only (with OpenMP), GPU-only (with CUDA), IHPv1 (OpenMP + CUDA), and IHPv2 (OpenMP + CUDA) implementations using exponential workloads on Desktop system. Single-precision computations. Total execution times noted in **bold**.

		Brightness		Chromaticity	
		Isotropic	Anisotropic	Isotropic	Anisotropic
OpenMP		**44.97**	**344.25**	**269.63**	**983.91**
CUDA		**6.58**	**59.19**	**65.12**	**226.07**
IHPv1	CPU	5.90	53.62	56.85	193.03
	GPU	6.05	53.86	57.10	197.51
	Copy	0.06	0.05	0.17	3.56
	Total	**6.11**	**55.05**	**57.41**	**200.54**
IHPv2	CPU	5.74	52.94	56.84	194.20
	GPU	6.10	54.16	57.42	197.31
	Copy	0.06	0.05	0.17	2.29
	Total	**6.21**	**54.74**	**57.61**	**198.94**

Table A.4: Execution time (in seconds) of CPU-only (with OpenMP), GPU-only (with CUDA), IHPv1 (OpenMP + CUDA), and IHPv2 (OpenMP + CUDA) implementations using linear workloads on Desktop system. Double-precision computations. Total execution times noted in **bold**.

		Brightness		Chromaticity	
		Isotropic	Anisotropic	Isotropic	Anisotropic
OpenMP		**23.85**	**173.86**	**136.95**	**494.76**
CUDA		**3.07**	**31.16**	**34.17**	**114.60**
IHPv1	CPU	3.05	28.90	31.22	102.98
	GPU	3.09	29.69	31.90	106.19
	Copy	0.12	0.92	1.05	4.51
	Total	**3.25**	**30.57**	**32.75**	**109.92**
IHPv2	CPU	3.02	29.61	32.01	105.97
	GPU	3.10	29.84	32.06	106.25
	Copy	0.06	0.05	0.17	1.47
	Total	**3.19**	**29.94**	**32.31**	**107.63**

Table A.5: Execution time (in seconds) of CPU-only (with OpenMP), GPU-only (with CUDA), IHPv1 (OpenMP + CUDA), and IHPv2 (OpenMP + CUDA) implementations using logarithmic workloads on Desktop system. Double-precision computations. Total execution times noted in **bold**.

		Brightness		Chromaticity	
		Isotropic	Anisotropic	Isotropic	Anisotropic
OpenMP		**214.48**	**1711.84**	**1335.90**	**4910.33**
CUDA		**3.08**	**294.33**	**322.82**	**1125.00**
IHPv1	CPU	2.66	194.90	194.32	682.31
	GPU	2.64	196.17	196.80	683.16
	Copy	0.09	0.81	0.18	0.18
	Total	**2.86**	**201.49**	**199.26**	**689.81**
IHPv2	CPU	2.64	193.68	194.59	680.20
	GPU	2.67	197.48	197.35	684.09
	Copy	0.10	0.05	1.71	1.21
	Total	**2.86**	**201.13**	**199.01**	**689.73**

Table A.6: Execution time (in seconds) of CPU-only (with OpenMP), GPU-only (with CUDA), IHPv1 (OpenMP + CUDA), and IHPv2 (OpenMP + CUDA) implementations using exponential workloads on Desktop system. Double-precision computations. Total execution times noted in **bold**.

		Brightness		Chromaticity	
		Isotropic	Anisotropic	Isotropic	Anisotropic
TBB		**7.03**	**72.76**	**70.89**	**246.86**
OpenCL		**4.98**	**61.21**	**54.51**	**134.59**
IHPv1	CPU	4.49	35.95	33.06	93.21
	GPU	4.74	37.85	35.43	98.25
	Copy	0.09	2.01	2.32	5.08
	Total	**4.85**	**40.44**	**37.93**	**103.55**
IHPv2	CPU	4.44	36.13	33.13	93.22
	GPU	4.69	37.98	35.58	98.66
	Copy	0.11	2.01	2.35	5.11
	Total	**4.80**	**40.63**	**38.00**	**103.75**

Table A.7: Execution time (in seconds) of CPU-only (with TBB), GPU-only (with OpenCL), IHPv1 (TBB + OpenCL), and IHPv2 (TBB + OpenCL) implementations using linear workloads on Laptop system. Single-precision computations. Total execution times noted in **bold**.

		Brightness		Chromaticity	
		Isotropic	Anisotropic	Isotropic	Anisotropic
TBB		**3.69**	**35.59**	**35.65**	**122.75**
OpenCL		**15.49**	**31.43**	**28.68**	**68.34**
IHPv1	CPU	3.49	23.34	22.32	61.20
	GPU	5.34	27.27	24.41	63.68
	Copy	0.66	2.33	1.80	2.71
	Total	**6.02**	**29.99**	**26.85**	**66.91**
IHPv2	CPU	3.31	23.62	22.33	61.38
	GPU	5.15	26.59	24.42	63.77
	Copy	0.66	1.83	1.80	2.47
	Total	**6.03**	**28.43**	**26.88**	**66.90**

Table A.8: Execution time (in seconds) of CPU-only (with TBB), GPU-only (with OpenCL), IHPv1 (TBB + OpenCL), and IHPv2 (TBB + OpenCL) implementations using logarithmic workloads on Laptop system. Single-precision computations. Total execution times noted in **bold**.

		Brightness		Chromaticity	
		Isotropic	Anisotropic	Isotropic	Anisotropic
TBB		**33.81**	**368.40**	**371.53**	**1235.94**
OpenCL		**136.13**	**299.02**	**262.19**	**520.04**
IHPv1	CPU	18.08	109.29	103.53	289.56
	GPU	21.71	111.26	105.27	296.04
	Copy	0.99	5.05	5.49	17.08
	Total	**26.41**	**139.54**	**128.94**	**357.97**
IHPv2	CPU	17.98	109.17	103.96	288.92
	GPU	21.57	111.54	105.17	296.23
	Copy	1.01	5.53	5.20	17.42
	Total	**26.10**	**137.64**	**127.33**	**353.54**

Table A.9: Execution time (in seconds) of CPU-only (with TBB), GPU-only (with OpenCL), IHPv1 (TBB + OpenCL), and IHPv2 (TBB + OpenCL) implementations using exponential workloads on Laptop system. Single-precision computations. Total execution times noted in **bold**.

Appendix B

Results on NUMA scheduling

Algorithm	BT.C	CG.C	EP.D	FT.C	IS.D	LU.C	MG.D	SP.C	UA.C
Baseline	41	21	73	9.1	26	58	80	36	44
Direct	67	49	73	15.9	35	50	326	112	85
Interleave	46	22	74	10.1	28	34	174	54	52
CRA	45	21	73	9.1	29	35	148	52	54
LBMA	44	21	73	9.1	28	37	135	47	50
IMAR^2	41	19	73	8.9	27	34	103	42	45
CIMAR	38	16	73	8.9	26	34	63	33	43
NIMAR	38	17	73	9.0	26	34	63	33	41
SMA	38	17	73	9.0	26	36	63	32	43
DyRMMA	38	20	71	8.0	24	53	70	35	41
LBMA+TMMA	44	21	73	8.8	28	33	138	47	51
IMAR^2+TMMA	41	19	73	9.0	27	37	103	42	46
CIMAR+TMMA	39	17	74	9.1	26	32	64	33	43
NIMAR+TMMA	38	18	73	8.8	26	31	66	33	42
SMA+TMMA	38	17	74	8.8	26	32	66	33	42
DyRMMA+TMMA	38	20	71	8.4	24	41	72	35	42
LBMA+LMMA	43	21	73	8.9	28	37	135	47	51
IMAR^2+LMMA	42	19	73	8.8	27	32	104	43	46
CIMAR+LMMA	39	16	73	8.8	26	33	65	34	42
NIMAR+LMMA	38	17	74	9.1	26	31	66	32	42
SMA+LMMA	38	17	73	8.8	26	32	65	33	41
DyRMMA+LMMA	39	18	71	8.3	24	46	72	36	41
RMMA	77	42	74	9.5	30	129	126	81	184
TMMA	41	20	74	9.0	27	50	75	35	43
LMMA	41	20	73	8.8	26	51	76	32	44

Table B.1: Execution times (in seconds) for Experiment Single in Server ct1. Lower is better.

Algorithm	BT.C	CG.C	EP.D	FT.C	IS.D	LU.C	MG.D	SP.C	UA.C
Baseline	23	7.68	18	8.29	7.20	44	59	24	40
Direct	61	20.12	18	20.93	31.83	62	367	127	99
Interleave	17	8.70	18	3.98	7.02	15	68	20	34
CRA	19	7.87	19	6.68	7.27	29	61	21	38
LBMA	20	8.26	18	6.49	7.31	39	58	21	36
IMAR^2	20	7.93	18	7.26	7.44	25	57	22	36
CIMAR	19	8.18	18	6.71	7.30	23	57	21	34
NIMAR	19	7.75	18	6.73	7.47	32	57	21	35
SMA	21	7.86	18	6.28	7.25	28	58	22	35
DyRMMA	19	9.15	18	6.89	7.37	25	64	22	39
LBMA+TMMA	19	7.71	18	6.70	7.28	27	59	21	36
IMAR^2+TMMA	19	7.82	18	7.25	7.40	26	57	21	36
CIMAR+TMMA	21	8.37	18	7.18	7.35	30	62	22	36
NIMAR+TMMA	18	7.77	18	5.93	7.30	28	60	21	36
SMA+TMMA	20	8.14	18	6.79	7.21	25	58	22	36
DyRMMA+TMMA	24	7.66	18	7.60	7.16	29	68	22	38
LBMA+LMMA	20	7.53	18	6.96	7.33	27	58	21	36
IMAR^2+LMMA	20	7.99	18	6.55	7.40	27	61	22	36
CIMAR+LMMA	21	7.84	18	6.51	7.22	27	63	21	36
NIMAR+LMMA	19	7.40	18	7.66	7.36	33	59	21	36
SMA+LMMA	20	7.86	18	7.08	7.29	34	59	23	36
DyRMMA+LMMA	19	7.62	18	6.65	7.29	36	66	21	38
RMMA	30	7.89	18	7.37	7.31	82	64	33	73
TMMA	23	8.07	19	7.63	7.36	43	60	24	41
LMMA	20	8.04	18	7.27	7.25	45	62	23	43

Table B.2: Execution times (in seconds) for Experiment Single in Server ct2. Lower is better.

Algorithm	BT.C	CG.C	LU.C	SP.C	Total	Accum.
Baseline	158.6	86.6	105.4	99.3	414.0	902.1
Direct	151.8	47.7	94.5	90.8	407.0	769.6
Interleave	168.6	100.8	111.4	118.6	430.0	999.9
CRA	185.1	108.5	123.1	132.2	451.0	1,098.5
LBMA	194.6	112.2	128.2	134.0	454.0	1,139.0
IMAR^2	167.7	85.9	110.7	108.1	423.0	952.1
CIMAR	152.3	49.9	104.9	83.1	400.0	783.5
NIMAR	156.3	57.7	107.3	89.8	403.0	822.6
SMA	154.1	50.7	108.2	83.8	403.0	791.7
DyRMMA	170.5	99.0	113.8	121.2	437.0	1,009.0
LBMA+TMMA	194.0	104.3	129.4	132.5	451.0	1,124.0
IMAR^2+TMMA	167.3	78.2	112.6	108.4	421.0	930.8
CIMAR+TMMA	153.3	50.1	106.6	83.9	404.0	789.6
NIMAR+TMMA	156.7	51.8	108.3	88.9	405.0	805.4
SMA+TMMA	155.6	50.2	108.3	87.9	402.0	802.1
DyRMMA+TMMA	170.5	96.3	113.2	121.1	437.0	1,001.9
LBMA+LMMA	190.6	110.7	130.5	132.9	452.0	1,128.0
IMAR^2+LMMA	164.8	79.1	110.7	110.0	429.0	940.7
CIMAR+LMMA	153.5	51.3	106.6	83.8	402.0	787.6
NIMAR+LMMA	156.0	53.7	108.6	85.6	403.0	809.6
SMA+LMMA	156.0	50.2	108.0	84.2	403.0	791.6
DyRMMA+LMMA	169.9	97.3	113.7	121.9	436.0	1,005.4
RMMA	196.2	102.8	151.9	146.6	458.0	1,194.4
TMMA	161.0	69.1	105.2	97.2	414.0	875.2
LMMA	160.3	83.3	105.9	102.5	417.0	903.4

Table B.3: Execution times (in seconds) for Experiment Interactive in Server ct1. Lower is better.

Algorithm	BT.C	CG.C	LU.C	SP.C	Total	Accum.
Baseline	54.3	26.9	45.2	45.3	204.0	340.9
Direct	100.4	32.8	1,282.3	110.4	1,440.0	3,052.7
Interleave	53.4	29.8	41.0	45.7	202.0	340.0
CRA	59.0	35.2	48.7	51.4	209.0	385.7
LBMA	57.3	23.2	50.9	47.7	205.0	362.1
IMAR^2	53.6	21.1	45.9	44.2	202.0	329.0
CIMAR	52.9	20.4	46.0	42.5	202.0	329.4
NIMAR	56.5	22.0	45.1	46.5	203.0	339.8
SMA	54.6	21.3	48.4	43.0	199.0	334.5
DyRMMA	57.8	34.2	45.4	52.1	210.0	376.9
LBMA+TMMA	59.7	22.6	55.3	48.2	206.0	363.2
IMAR^2+TMMA	53.5	20.6	48.2	43.7	199.0	333.6
CIMAR+TMMA	56.8	21.7	48.2	44.7	201.0	347.3
NIMAR+TMMA	55.8	24.6	45.1	46.5	203.0	339.9
SMA+TMMA	54.1	20.6	45.9	41.4	197.0	324.4
DyRMMA+TMMA	59.4	34.9	48.1	52.4	209.0	391.4
LBMA+LMMA	56.6	21.8	48.1	47.3	205.0	345.9
IMAR^2+LMMA	54.3	22.3	49.5	44.3	201.0	347.4
CIMAR+LMMA	53.3	20.3	45.8	43.8	202.0	324.3
NIMAR+LMMA	55.0	24.2	44.8	44.6	200.0	336.8
SMA+LMMA	54.5	19.4	44.0	41.0	197.0	320.4
DyRMMA+LMMA	58.0	34.5	47.1	51.5	209.0	380.6
RMMA	63.0	34.6	69.8	65.1	220.0	467.3
TMMA	54.2	27.3	46.0	46.2	205.0	350.3
LMMA	54.9	26.1	45.8	44.5	203.0	341.4

Table B.4: Execution times (in seconds) for Experiment Interactive in Server ct2. Lower is better.

Algorithm	BT.C	CG.C	EP.D	FT.C	IS.D	LU.C	MG.D	SP.C	UA.C	Total	Accum.
Baseline	145	75	292	31	89	98	240	111	139	1,325	4,888
Direct	133	44	291	29	54	85	199	90	126	1,129	4,220
Interleave	169	122	296	35	145	116	359	160	181	1,732	6,343
CRA	150	78	293	31	110	105	258	113	142	1,407	5,197
LBMA	146	77	291	30	100	107	240	110	149	1,416	5,052
IMAR^2	147	82	292	31	101	104	234	111	160	1,392	5,072
CIMAR	144	64	293	31	94	96	185	96	123	1,284	4,587
NIMAR	139	53	292	30	94	97	171	85	120	1,190	4,344
SMA	139	52	292	30	95	96	166	83	119	1,188	4,311
DyRMMA	157	105	292	35	124	109	323	131	156	1,554	5,753
LBMA+TMMA	147	75	293	31	99	104	246	111	157	1,413	5,078
IMAR^2+TMMA	144	76	309	31	102	99	242	106	139	1,392	5,027
CIMAR+TMMA	145	61	297	31	94	97	179	92	128	1,292	4,477
NIMAR+TMMA	139	53	292	30	92	98	174	85	122	1,199	4,384
SMA+TMMA	139	51	292	30	96	96	169	84	122	1,182	4,337
DyRMMA+TMMA	156	100	293	35	124	110	330	131	158	1,566	5,784
LBMA+LMMA	139	75	317	31	101	99	238	110	148	1,413	5,059
IMAR^2+LMMA	147	74	292	32	106	104	254	111	143	1,401	5,101
CIMAR+LMMA	144	52	293	30	94	95	166	87	118	1,214	4,283
NIMAR+LMMA	138	53	292	30	97	98	173	84	121	1,196	4,371
SMA+LMMA	140	51	292	30	96	98	160	84	121	1,183	4,312
DyRMMA+LMMA	156	99	292	35	123	109	325	130	156	1,549	5,721
RMMA	194	103	294	34	98	128	272	180	192	1,611	6,013
TMMA	143	61	292	31	92	97	240	110	143	1,327	4,850
LMMA	145	69	292	31	90	96	237	106	144	1,314	4,848

Table B.5: Execution times (in seconds) for Experiment Queue in Server ct1. Lower is better.

Algorithm	BT.C	CG.C	EP.D	FT.C	IS.D	LU.C	MG.D	SP.C	UA.C	Total	Accum.
Baseline	103	61	137	24	46	82	274	113	180	1,138	8,174
Direct	92	34	135	20	29	69	260	100	158	1,025	7,216
Interleave	117	89	142	26	62	94	325	142	215	1,332	9,713
CRA	109	67	137	23	52	88	276	118	187	1,195	8,480
LBMA	98	55	143	22	47	82	234	108	174	1,112	7,757
IMAR^2	116	75	136	24	57	92	304	129	213	1,241	9,183
CIMAR	102	51	138	21	50	86	232	96	187	1,061	7,805
NIMAR	100	48	136	21	47	78	219	101	171	1,058	7,387
SMA	104	56	138	22	48	84	242	109	179	1,120	7,893
DyRMMA	103	51	136	21	48	88	234	103	185	1,068	7,765
LBMA+TMMA	99	48	134	21	46	81	213	101	172	1,021	7,378
IMAR^2+TMMA	117	69	136	23	58	92	305	131	218	1,253	9,238
CIMAR+TMMA	104	51	134	21	48	84	231	99	187	1,062	7,720
NIMAR+TMMA	101	49	137	21	45	83	221	101	174	1,038	7,455
SMA+TMMA	105	60	139	22	48	83	260	112	184	1,160	8,158
DyRMMA+TMMA	105	50	137	22	47	85	228	98	183	1,064	7,682
LBMA+LMMA	100	47	138	22	47	82	217	102	172	1,037	7,470
IMAR^2+LMMA	116	72	138	23	57	93	304	130	218	1,247	9,234
CIMAR+LMMA	104	49	134	21	49	85	231	100	186	1,062	7,689
NIMAR+LMMA	98	48	138	21	45	83	225	99	174	1,034	7,465
SMA+LMMA	105	55	141	24	48	84	255	110	181	1,124	8,024
DyRMMA+LMMA	104	48	136	21	48	85	228	99	187	1,055	7,697
RMMA	113	63	135	24	50	92	290	126	193	1,195	8,760
TMMA	103	55	139	24	48	85	274	111	179	1,137	8,163
LMMA	105	56	135	24	49	83	268	111	177	1,145	8,123

Table B.6: Execution times (in seconds) for Experiment Queue in Server ct2. Lower is better.

Figure B.1: Execution time (in seconds) for different amount of remote pages—over a total of 1 000 000 pages—and operations to migrate a page, with 100 % chances of migration. 1 thread per worker group.

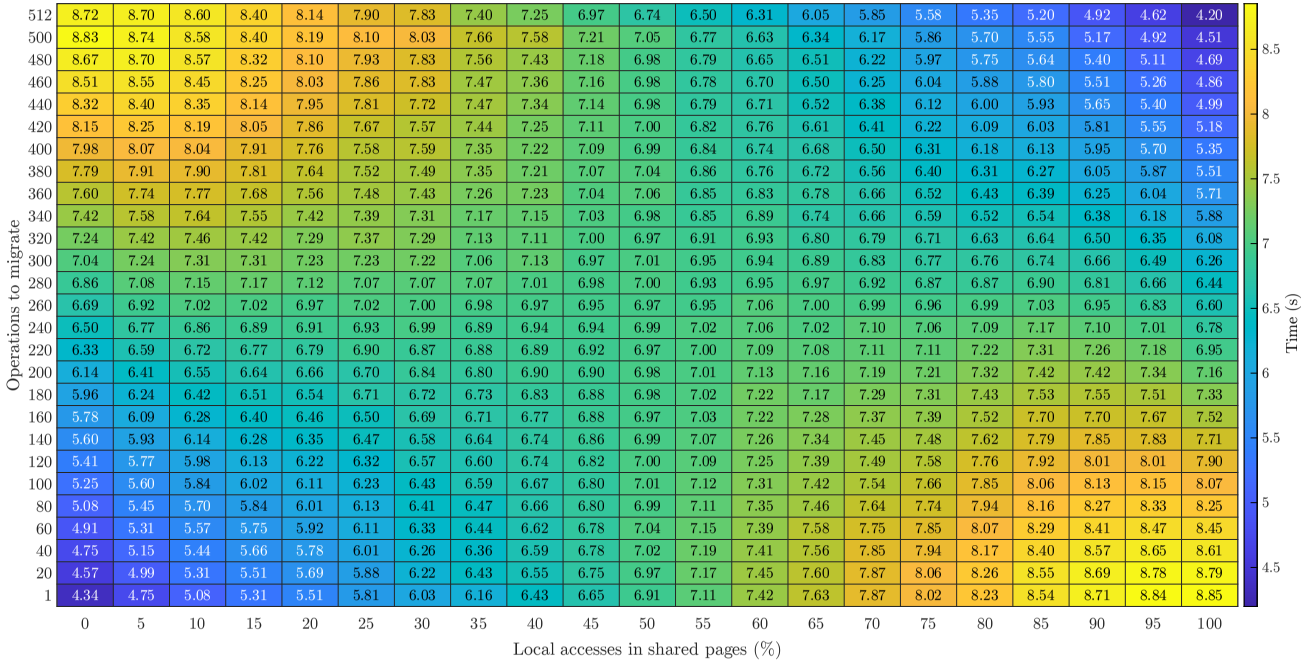


Figure B.2: Execution time (in seconds) for different amount of remote pages—over a total of 1 000 000 pages—and operations to migrate a page, with 100 % chances of migration. 10 threads per worker group.

Appendix C

Derived publications

Presented below is a compendium of scholarly publications derived from the doctoral dissertation:

- Articles in peer-reviewed journals:
  - R. Laso, O. G. Lorenzo, J. C. Cabaleiro, T. F. Pena, J. Á. Lorenzo, and F. F. Rivera, "CIMAR, NIMAR, and LMMA: Novel algorithms for thread and memory migrations in user space on NUMA systems using hardware counters", *Future Generation Computer Systems*, vol. 129, pp. 18–32, 2022, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2021.11.008.
    Impact factor (JCR 2021): 7.307.
    Category: Computer Science, Theory & Methods.
    Rank: 10/110.
    Contributions: Development of the algorithms, implementation of the software, design of the experiments, analysis of the results and writing of the article.
  - R. Laso, J. C. Cabaleiro, F. F. Rivera, M. C. Muñiz, and J. A. Álvarez-Dios, "IHP: A dynamic heterogeneous parallel scheme for iterative or time-step methods—image denoising as case study", *The Journal of Supercomputing*, vol. 77, no. 1, pp. 95–110, Jan. 2021. DOI: https://doi.org/10.1007/s11227-020-03260-8.
    Impact factor (JCR 2021): 2.557.
    Category: Computer Science, Theory & Methods.
    Rank: 43/110.
    Contributions: Development of the algorithms regarding workload balance, implementation of the software, design of the experiments, analysis of the results and writing of the article.

- R. Laso, O. G. Lorenzo, F. F. Rivera, J. C. Cabaleiro, T. F. Pena, and J. A. Lorenzo, "LBMA and IMAR^2: Weighted lottery based migration strategies for NUMA multiprocessing servers", *Concurrency and Computation: Practice and Experience*, vol. 33, no. 11, e5950, 2021. DOI: https://doi.org/10.1002/cpe.5950.
  Impact factor (JCR 2021): 1.831.
  Category: Computer Science, Theory & Methods.
  Rank: 58/110.
  Contributions: Development of the algorithms, implementation of the software, design of the experiments, analysis of the results and writing of the article.
- R. Laso, J. C. Cabaleiro, F. F. Rivera, M. C. Muñiz, and J. A. Álvarez-Dios, "Load balanced heterogeneous parallelism for finite difference problems on image denoising", *Computational and Mathematical Methods*, vol. 3, no. 3, e1089, 2021. DOI: https://doi.org/10.1002/cmm4.1089.
  Impact factor (JCI 2021): 0.50.
  Category: Applied Mathematics.
  Rank: 217/317.
  Contributions: Development of the algorithms regarding workload balance, implementation of the software, design of the experiments, analysis of the results and writing of the article.

- Articles published in international conferences:
  - R. Laso, F. F. Rivera, and J. C. Cabaleiro, "Influence of architectural features of the SNC-4 mode of the Intel Xeon Phi KNL on matrix multiplication", in *Computational Science – ICCS 2019*, J. M. F. Rodrigues, P. J. S. Cardoso, J. Monteiro, R. Lam, V. V. Krzhizhanovskaya, M. H. Lees, J. J. Dongarra, and P. M. Sloot, Eds., Cham: Springer International Publishing, 2019, pp. 483–490, ISBN: 978-3-030-22750-0. DOI: https://doi.org/10.1007/978-3-030-22750-0_41.
    Class: 3 (CORE A).
    Contributions: Implementation of the software, design of the experiments, analysis of the results and writing of the article.
  - O. García Lorenzo, R. Laso Rodríguez, T. Fernández Pena, J. C. Cabaleiro Domínguez, F. Fernández Rivera, and J. Á. Lorenzo del Castillo, "A new hardware counters based thread migration strategy for NUMA systems", in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, E. Deelman, J. Dongarra, and K. Karczewski, Eds., Cham: Springer International Publishing, 2020, pp. 205–216, ISBN: 978-3-030-43222-5. DOI: https://doi.org/10.1007/978-3-030-43222-5_18.
    Class: W.

Contributions: Development of the algorithms (partially), implementation of the software, design of the experiments (partially), analysis of the results and writing of the article (partially).

- Articles published in national conferences:
  - R. Laso, F. F. Rivera, and J. C. Cabaleiro, "Producto matricial en el Intel Xeon Phi KNL en el modo Sub-NUMA Clustering 4", in *Jornadas SARTECO 2018*, 2018, pp. 129–136.
    Contributions: Implementation of the software, design of the experiments, analysis of the results and writing of the article.

Appendix D

Copyright and permissions

- R. Laso, F. F. Rivera, and J. C. Cabaleiro, "Influence of architectural features of the SNC-4 mode of the Intel Xeon Phi KNL on matrix multiplication", in *Computational Science – ICCS 2019*, J. M. F. Rodrigues, P. J. S. Cardoso, J. Monteiro, R. Lam, V. V. Krzhizhanovskaya, M. H. Lees, J. J. Dongarra, and P. M. Sloot, Eds., Cham: Springer International Publishing, 2019, pp. 483–490, ISBN: 978-3-030-22750-0. DOI: https://doi.org/10.1007/978-3-030-22750-0_41.

> Springer Nature Book and Journal Authors have the right to reuse the Version of Record, in whole or in part, in their own thesis. Additionally, they may reproduce and make available their thesis, including Springer Nature content, as required by their awarding academic institution. Authors must properly cite the published work in their thesis according to current citation standards and include the following acknowledgement: *'Reproduced with permission from Springer Nature'*. Copyright terms: https://www.springernature.com/gp/partners/rights-permissions-third-party-distribution.

- O. García Lorenzo, R. Laso Rodríguez, T. Fernández Pena, J. C. Cabaleiro Domínguez, F. Fernández Rivera, and J. Á. Lorenzo del Castillo, "A new hardware counters based thread migration strategy for NUMA systems", in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, E. Deelman, J. Dongarra, and K. Karczewski, Eds., Cham: Springer International Publishing, 2020, pp. 205–216, ISBN: 978-3-030-43222-5. DOI: https://doi.org/10.1007/978-3-030-43222-5_18.

- R. Laso, J. C. Cabaleiro, F. F. Rivera, M. C. Muñiz, and J. A. Álvarez-Dios, "Load balanced heterogeneous parallelism for finite difference problems on image denoising", *Computational and Mathematical Methods*, vol. 3, no. 3, e1089, 2021. DOI: https://doi.org/10.1002/cmm4.1089.

- R. Laso, J. C. Cabaleiro, F. F. Rivera, M. C. Muñiz, and J. A. Álvarez-Dios, "IHP: A dynamic heterogeneous parallel scheme for iterative or time-step methods—image denoising as case study", *The Journal of Supercomputing*, vol. 77, no. 1, pp. 95–110, Jan. 2021. DOI: https://doi.org/10.1007/s11227-020-03260-8.

- R. Laso, O. G. Lorenzo, F. F. Rivera, J. C. Cabaleiro, T. F. Pena, and J. A. Lorenzo, "LBMA and IMAR^2: Weighted lottery based migration strategies for NUMA multiprocessing servers", *Concurrency and Computation: Practice and*

Experience, vol. 33, no. 11, e5950, 2021. DOI: https://doi.org/10.1002/cpe.5950.

> "John Wiley and Sons" offers the reuse of its content for a thesis or dissertation free of charge according to their copyright policy: https://www.wiley.com/en-us/network/publishing/research-publishing/trending-stories/how-to-clear-permissions-for-a-thesis-or-dissertation.

- R. Laso, O. G. Lorenzo, J. C. Cabaleiro, T. F. Pena, J. Á. Lorenzo, and F. F. Rivera, "CIMAR, NIMAR, and LMMA: Novel algorithms for thread and memory migrations in user space on NUMA systems using hardware counters", *Future Generation Computer Systems*, vol. 129, pp. 18–32, 2022, ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2021.11.008.

> Please note that, as the author of this Elsevier article, you retain the right to include it in a thesis or dissertation, provided it is not published commercially. Permission is not required, but please ensure that you reference the journal as the original source. For more information on this and on your other retained rights, please visit: https://www.elsevier.com/about/our-business/policies/copyright#Author-rights

SPRINGER NATURE LICENSE
TERMS AND CONDITIONS

Mar 02, 2023

This Agreement between Mr. Ruben Laso Rodriguez ("You") and Springer Nature ("Springer Nature") consists of your license details and the terms and conditions provided by Springer Nature and Copyright Clearance Center.

License Number	5500670739553
License date	Mar 02, 2023
Licensed Content Publisher	Springer Nature
Licensed Content Publication	Springer eBook
Licensed Content Title	Influence of Architectural Features of the SNC-4 Mode of the Intel Xeon Phi KNL on Matrix Multiplication
Licensed Content Author	Ruben Laso, Francisco F. Rivera, José Carlos Cabaleiro
Licensed Content Date	Jan 1, 2019
Type of Use	Thesis/Dissertation
Requestor type	academic/university or research institute
Format	print and electronic
Portion	full article/chapter
Will you be translating?	no
Circulation/distribution	1 - 29
Author of this Springer Nature content	yes

SPRINGER NATURE LICENSE
TERMS AND CONDITIONS

Mar 02, 2023

This Agreement between Mr. Ruben Laso Rodriguez ("You") and Springer Nature ("Springer Nature") consists of your license details and the terms and conditions provided by Springer Nature and Copyright Clearance Center.

License Number	5500670777278
License date	Mar 02, 2023
Licensed Content Publisher	Springer Nature
Licensed Content Publication	Springer eBook
Licensed Content Title	A New Hardware Counters Based Thread Migration Strategy for NUMA Systems
Licensed Content Author	Oscar García Lorenzo, Rubén Laso Rodríguez, Tomás Fernández Pena et al
Licensed Content Date	Jan 1, 2020
Type of Use	Thesis/Dissertation
Requestor type	academic/university or research institute
Format	print and electronic
Portion	full article/chapter
Will you be translating?	no
Circulation/distribution	1 - 29

Author of this Springer Nature content yes

JOHN WILEY AND SONS LICENSE
TERMS AND CONDITIONS

Mar 01, 2023

This Agreement between Mr. Ruben Laso Rodriguez ("You") and John Wiley and Sons ("John Wiley and Sons") consists of your license details and the terms and conditions provided by John Wiley and Sons and Copyright Clearance Center.

License Number	5500070727613
License date	Mar 01, 2023
Licensed Content Publisher	John Wiley and Sons
Licensed Content Publication	COMPUTATIONAL AND MATHEMATICAL METHODS
Licensed Content Title	Load balanced heterogeneous parallelism for finite difference problems on image denoising
Licensed Content Author	José A. Álvarez-Dios, M. Carmen Muñiz, Francisco F. Rivera, et al
Licensed Content Date	Mar 13, 2020
Licensed Content Volume	3
Licensed Content Issue	3
Licensed Content Pages	13
Type of use	Dissertation/Thesis
Requestor type	Author of this Wiley article
Format	Print and electronic

SPRINGER NATURE LICENSE
TERMS AND CONDITIONS

Mar 01, 2023

This Agreement between Mr. Ruben Laso Rodriguez ("You") and Springer Nature ("Springer Nature") consists of your license details and the terms and conditions provided by Springer Nature and Copyright Clearance Center.

License Number	5500070848764
License date	Mar 01, 2023
Licensed Content Publisher	Springer Nature
Licensed Content Publication	Journal of Supercomputing, The
Licensed Content Title	IHP: a dynamic heterogeneous parallel scheme for iterative or time-step methods—image denoising as case study
Licensed Content Author	Ruben Laso et al
Licensed Content Date	Mar 26, 2020
Type of Use	Thesis/Dissertation
Requestor type	academic/university or research institute
Format	print and electronic
Portion	full article/chapter
Will you be translating?	no
Circulation/distribution	1 - 29

JOHN WILEY AND SONS LICENSE
TERMS AND CONDITIONS

Mar 01, 2023

This Agreement between Mr. Ruben Laso Rodriguez ("You") and John Wiley and Sons ("John Wiley and Sons") consists of your license details and the terms and conditions provided by John Wiley and Sons and Copyright Clearance Center.

License Number	5500070965645
License date	Mar 01, 2023
Licensed Content Publisher	John Wiley and Sons
Licensed Content Publication	CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Licensed Content Title	LBMA and IMAR2: Weighted lottery based migration strategies for NUMA multiprocessing servers
Licensed Content Author	R. Laso, O. G. Lorenzo, F. F. Rivera, et al
Licensed Content Date	Aug 17, 2020
Licensed Content Volume	33
Licensed Content Issue	11
Licensed Content Pages	14
Type of use	Dissertation/Thesis
Requestor type	Author of this Wiley article
Format	Print and electronic

CIMAR, NIMAR, and LMMA: Novel algorithms for thread and memory migrations in user space on NUMA systems using hardware counters

Author:
Ruben Laso,Oscar G. Lorenzo,José C. Cabaleiro,Tomás F. Pena,Juan Ángel Lorenzo,Francisco F. Rivera

Publication: Future Generation Computer Systems

Publisher: Elsevier

Date: April 2022

This thesis faces the challenges on dynamic workload optimisation and workload balancing in two different problems: in conventional systems using heterogeneous (CPU and GPU) parallelism, and in NUMA systems.
On one hand, a library named IHP is proposed. Dynamically, the performance of the CPU and GPU is evaluated so the workload is divided accordingly. Results show that execution times can be improved between 3% and 55% depending on the code and the performance of the computing units.
On the other hand, a tool for the migration of threads and memory pages in NUMA systems has been developed. This tool incorporates several algorithms that, considering performance measurements, decide whether a migration is required. Experiments show that performance can be improved by up to 47%, particularly in multi-tasking scenarios.