Library-based solutions for algorithms with complex patterns of parallelism

Author: Carlos Hugo González Vázquez

PhD Thesis - 2015

Advisor: Basilio Bernardo Fraguela Rodríguez

Departamento de Electrónica e Sistemas

UNIVERSIDADE DA CORUÑA

Dr. Basilio Bernardo Fraguela Rodríguez

Profesor Titular de Universidade

Dpto. de Electrónica e Sistemas da Universidade da Coruña

CERTIFICA

Que a memoria titulada “Library-based solutions for algorithms with complex patterns of parallelism” foi realizada por D. Carlos Hugo González Vázquez baixo a miña dirección no Departamento de Electrónica e Sistemas da Universidade da Coruña e conclúe a Tese de Doutoramento que presenta para a obtención do título de Doutor pola Universidade da Coruña coa Mención de Doutor Internacional.

Na Coruña, o …… de …….. de …….

Asdo.: Basilio Bernardo Fraguela Rodríguez

Director da Tese de Doutoramento

Asdo.: Carlos Hugo González Vázquez

Autor da Tese de Doutoramento

Resumo

Coa chegada dos procesadores multinúcleo e a caída do crecemento da capacidade de procesamento por núcleo en cada nova xeración, a paralelización é cada vez máis crítica para mellorar o rendemento de todo tipo de aplicacións. Ademais, hai un bo coñecemento e soporte dos patróns de paralelismo máis sinxelos, mais non sendo así para patróns complexos e irregulares, cuxa paralelización require ben ferramentas de baixo nivel que afectan negativamente á produtividade, ben solucións transaccionais con requisitos específicos de hardware ou que implican grandes sobrecostes. O aumento do número de aplicacións que exhiben estes patróns complexos fai que este sexa un problema con importancia crecente. Esta tese trata de mellorar a comprensión e o soporte de tres tipos de patróns complexos mediante a identificación de abstraccións e semánticas claras que axuden a súa paralelización en entornos de memoria compartida. O enfoque elixido foi a creación de librarías, xa que facilitan a reutilización de código, reducen os requisitos do compilador, e teñen unha curva de aprendizaxe relativamente curta. A linguaxe empregada para a implementación é C++, pois proporciona un bo rendemento e capacidade para expresar as abstraccións necesarias. Os exemplos e avaliacións nesta tese mostran que as nosas propostas permiten expresar de xeito elegante as aplicacións que presentan estes patróns, mellorando a súa programabilidade ao tempo que proporcionan un rendemento similar ou superior ao de outras solucións existentes.

Abstract

With the arrival of multi-core processors and the reduction in the growth rate of the processing power per core in each new generation, parallelization is becoming increasingly critical to improve the performance of every kind of application. Also, while simple patterns of parallelism are well understood and supported, this is not the case for complex and irregular patterns, whose parallelization requires either low level tools that hurt programmers' productivity or transactional based approaches that need specific hardware or imply potentially large overheads. This is becoming an increasingly important problem as the number of applications that exhibit these latter patterns is steadily growing. This thesis tries to better understand and support three kinds of complex patterns through the identification of abstractions and clear semantics that help bring structure to them and the development of libraries based on our observations that facilitate their parallelization in shared memory environments. The library approach was chosen given its advantages for code reuse, reduced compiler requirements, and relatively short learning curve. The implementation language selected being C++ due to its good performance and capability to express abstractions. The examples and evaluations in this thesis show that our proposals allow to elegantly express the applications that present these patterns, improving their programmability while providing similar or even better performance than existing approaches.

Resumen

Con la llegada de los procesadores multinúcleo y la caída del crecimiento de la capacidad de procesamiento por núcleo en cada nueva generación, la paralelización es cada vez más crítica para mejorar el rendimiento de todo tipo de aplicaciones. Por otra parte, si bien hay un buen conocimiento y soporte de los patrones de paralelismo más sencillos, esto no es así para los patrones complejos e irregulares, cuya paralelización requiere o bien herramientas de bajo nivel que afectan negativamente a la productividad, o bien soluciones transaccionales con requisitos específicos de hardware o que implican grandes sobrecostes. El aumento del número de aplicaciones que exhiben estos patrones complejos hace que este sea un problema con importancia creciente. Esta tesis trata de mejorar la comprensión y el soporte de tres tipos de patrones complejos, mediante la identificación de abstracciones y semánticas claras que ayuden su paralelización en entornos de memoria compartida. El enfoque elegido fue la creación de librerías, ya que facilitan la reutilización de código, reducen los requisitos del compilador, y tienen una curva de aprendizaje relativamente corta. El lenguaje empleado para la implementación es C++, pues proporciona un buen rendimiento y capacidad para expresar las abstracciones necesarias. Los ejemplos y evaluaciones en esta tesis muestran que nuestras propuestas permiten expresar de manera elegante las aplicaciones que presentan estos patrones, mejorando su programabilidad al tiempo que proporcionan un rendimiento similar o superior al de otras soluciones existentes.

Prólogo

No campo da arquitectura de computadores véñense dando importantes e continuos avances nas últimas décadas. Desde o fito da invención do microprocesador 4004 de Intel, houbo unha progresión tanto en métodos de fabricación como en deseño de microarquitectura que produciu unha mellora continua no rendemento dos computadores. Unha das características máis visíbeis é o aumento da frecuencia de reloxo, que permite a un procesador executar máis instrucións na mesma cantidade de tempo, sen necesidade de modificar o código do programa nin de empregar optimizacións epeciais á hora de compilar. Pero a mellora dos deseños dos microprocesadores tamén permite aumentos de rendemento, agora si cunha programación mellor enfocada ou compiladores máis intelixentes. Os procesadores modernos contan con unidades de procesamento replicadas (ALUs, acceso a memoria, etc.) que permiten executar varias instrucións simultaneamente, co requerimento de estaren correctamente ordenadas e non se produciren conflitos. A combinación de soporte hardware e compiladores avanzados permite que os programas desenvolvidos cumpran estas condicións con relativa facilidade.

Porén, a mellora de frecuencia e deseño de microarquitectura está a chocar na actualidade coa limitación física do tamaño mínimo co que se pode fabricar un transistor. Isto impide aumentar as frecuencias de reloxo, alén de introducir novos problemas no deseño de microprocesadores. Isto levou ao crecemento do interese e a investigación en sistemas multiprocesador e multinúcleo (varios procesadores no mesmo chip). Estes sistemas permiten a execución de múltiples programas simultaneamente, e aínda que adoitaban estar reservados para entornos con grandes necesidades de rendemento ou throughtput, a día de hoxe atópanse mesmo en computadores persoais de calquera gama, ou dispositivos móbeis ou embebidos. É por iso que é fundamental mellorar a programabilidade nestes sistemas, pois actualmen-

te as ferramentas de programación para entornos multiprocesador son demasiado complexas por un lado, ou de aplicabilidade limitada por outro.

Nesta tese preséntanse unhas librarías para a linguaxe de programación C++, amplamente extendida, que permiten paralelizar con facilidade tres tipos de problema moi comúns, e suficientemente configurábeis para permitir o seu uso en grande variedade de circunstancias. Os tres patróns cubertos son:

Divide-e-vencerás Moitos problemas poden resolverse subdividíndoos en problemas máis pequenos. Estes subproblemas son con frecuencia resolúbeis máis facilmente, e aliás son independentes entre eles, de xeito que poden resolverse en paralelo.

Paralelismo de datos amorfo Cando a estrutura de datos empregada para a resolución dun problema é irregular resulta complicada a súa distribución entre diversas unidades de procesamento. Isto é debido a que unha estrutura irregular non ten un patrón de almacenamento definido. O exemplo paradigmático de estrutura irregular é o grafo, onde cada nodo é creado e enlazado con outros nodos dinamicamente, e durante a execución o número de nodos e as súas conexións pode variar. A problemática da paralelización de algoritmos que operan sobre sobre estruturas deste tipo tamén ven dada a miúdo pola imposibilidade de determinar a priori a independencia das tarefas a realizar e/ou a distinta carga computacional destas tarefas.

Pero unha análise coidadosa deste tipo de algoritmos desvela que adoitan axustarse a un patrón de procesamento, onde en cada momento hai un subconxunto de nodos activos que precisan un procesamento. Se os conxuntos formados por cada un destes nodos e os seus veciños non se intersecan, estes procesamentos pódense facer en paralelo, feito que explota a nosa libraría

Tarefas superescalares Noutros casos, un programa está composto por tarefas que serían candidatas a se executar paralelamente. Pero cando estas tarefas traballaren sobre datos comúns, deben ser ordenadas para non se produciren conflitos, só se executaren en paralelo aquelas que foren independentes en cada momento, e esperando á finalización daquelas tarefas que sexa preciso para manteren a semántica secuencial do programa. Estas tarefas pódense por tanto executar fora de orde, coa condición de se manter a semántica secuencial

do programa, dun xeito similar a como executan as instruccións os procesadores superescalares modernos, do que procede o termo tarefa superescalar comunmente usado na bibliografía.

Nesta tese desenvolveronse librarías orientadas a facilitar a paralelización de cada un dos tipos de problemas expostos. Estas librarías están desenvolvidas para C++, por ser esta unha das linguaxes de programación de uso máis extendido, tanto para aplicacións científicas en entornos de computación de altas prestacións, como para aplicacións de escritorio, videoxogos ou entornos de traballo. Orientando as librarías a unha linguaxe como C++ garantimos que poderán ser empregada en numerosos ámbitos.

Como base para o noso traballo usamos a API de baixo nivel das Intel Threading Building Blocks. Esta ferramenta proporciona unha serie de construcións de alto nivel que permiten paralelizar bucles con un esforzo relativamente limitado. Para alén, conta cunha interface de baixo nivel que se pode utilizar para realizar programas paralelos baseados en tarefas. Gracias ao emprego deste recurso, puidemos centrar o noso traballo en obter unha interface axeitada e implementar os mecanismos presentes nos patróns descritos anteriormente, mentres aproveitabamos a potencia do planificador de tarefas presente na libraría de Intel.

Metodoloxía de traballo

Para realizar esta tese seguiuse unha metodoloxía espiral coa que se puidesen aproveitar os coñecementos adquiridos en cada fase ou ben para as posteriores ou ben para refinamento das anteriores.

A tese conta con catro grandes bloques (B), con obxectivos (O) e tarefas (T):

B1.- Estudo de algoritmos.

- O1.- Atopar patróns comúns a diversos algoritmos.
  - T1.- Búsqueda de suites de benchmarks con uso real.
  - T2.- Análise das estructuras de datos.

T3.- Análise dos fluxos de execución.

O2.- Definición formal dos patróns.

T1.- Estabelecemento das características necesarias das estruturas de datos.

T2.- Definición das compoñentes dos programas.

T3.- Definición das interfaces necesarias.

O3.- Deseño de alto nivel da libraría.

T1.- Elección do entorno.

T2.- Estabelecemento dos módulos necesarios.

B2.- Estudo do estado da arte.

O1.- Búsqueda de solucións existentes.

T1.- Estudo da bibliografía.

T2.- Probas das solucións existentes aplicábeis.

T3.- Análise de vantaxes e desvantaxes.

B3.- Deseño e implementación dos módulos da libraría.

O1.- Patrón divide e vencerás.

T1.- Deseño do módulo.

T2.- Implementación do módulo.

T3.- Comprobación de corrección.

O2.- Patrón paralelismo de datos amorfo.

T1.- Deseño do módulo.

T2.- Implementación do módulo.

T3.- Comprobación de corrección.

O3.- Patrón tarefas dependentes.

T1.- Deseño do módulo.

T2.- Implementación do módulo.

T3.- Comprobación de corrección.

B4.- Análise de resultados.

O1.- Análise de programabilidade.

T1.- Medición de indicadores.

T2.- Comparación con alternativas.

O2.- Análise de rendemento.

T1.- Medición de tempos.

T2.- Medición de escalabilidade.

T3.- Comparación con alternativas.

Medios

Para a elaboración da tese empregáronse os medios detallados a continuación:

- Material de traballo e financiamento económico proporcionados polo Grupo de Arquitectura de Computadores da Universidade da Coruña e o Ministerio de Educación (bolsa predoutoral FPU AP2009-4752).
- Redes nas que se enmarca a tese:
  - Red Gallega de Computación de Altas Prestaciones.
  - Red Gallega de Computación de Altas Prestaciones II.
  - Red Mathematica Consulting & Computing de Galicia II.
  - High-Performance Embedded Architectures and Compilers Network of Excellence, HiPEAC2 NoE (ref. ICT-217068).
  - High-Performance Embedded Architectures and Compilers Network of Excellence, HiPEAC3 NoE (ref. ICT-287759).
  - Network for Sustainable Ultrascale Computing (NESUS). ICT COST Action IC0805.
  - Open European Network for High Performance Computing on Complex Environments (ComplexHPC). ICT COST Action IC0805
- Proxectos de investigación que financiaron esta tese:

- Soporte Hardware y Software para Computación de Altas Prestaciones (Ministerio de Economía y Competitividad, TIN2007-67537-C03-02)
- Mellora da programabilidade e do rendemento nas novas xeracións de computadores baseados en procesadores multinúcleo (Xunta de Galicia, INCITE08PXIB105161PR).
- Architectures, Systems and Tools for High Performance Computing (Ministerio de Economía y Competitividad, TIN2010-16735).
- Consolidación y Estructuración de Unidades de Investigación Competitivas ref. 2010/6: Grupo de Arquitectura de Computadores de la Universidad de A Coruña (Xunta de Galicia, UDC/GI-000265).
- Consolidación y Estructuración de Unidades de Investigación Competitivas: Grupo de Arquitectura de Computadores de la Universidad de A Coruña (Xunta de Galicia, GRC2013-055).
- Nuevos desafíos en la computación de altas prestaciones: Desde arquitecturas hasta aplicaciones. (Ministerio de Economía y Competitividad, TIN2013-42148-P).

- Clúster *pluton* do Grupo de Arquitectura de Computadores da Universidade da Coruña. Nodos con dous procesadores Intel Xeon E5-2660 Sandy Bridge-EP (16 núcleos por nodo) e 64 GB de RAM DDR3 a 1600 Mhz.

- Clúster *Finisterrae* do Centro de Supercomputación de Galicia (CESGA). Nodos HP Integrity rx7640 con 16 núcleos Itanium Montvale e 128 GB de RAM.

- Clúster *SVG* do Centro de Supercomputación de Galicia (CESGA). Nodos HP ProLiant SL165z G7 con dous procesadores AMD Opteron Processor 6174 e 64 GB de RAM.

- Estancia de 3 meses no grupo ISS do Prof. Pingali na University of Texas at Austin.

Conclusións

Durante décadas, a comunidade científica centrouse principalmente na paralelización de códigos con fluxos de control, estructuras e patróns de acceso regulares, xa que é no que se basean as aplicacións científicas e de enxeñaría para as que o procesamento paralelo era utilizado case en exclusiva até a aparición de procesadores multinúcleo, que extendeu o interese na paralelización de calquera tipo de aplicación. Como resultado, mentres as aplicacións con paralelismo de datos regular son ben entendidas e soportadas, algoritmos que se describen mellor en termos de patróns de paralelismo máis complexos en moitas ocasións necesitan que as programadoras recorran á parelelización manual empregando ou ben ferramentas de baixo nivel, o cal é susceptíbel a erros e custoso, ben solucións transaccionais con requisitos específicos de hardware ou que implican grandes sobrecostes de rendemento. Esta tese é un intento de entender mellor algúns destes problemas de prover ferramentas que melloren a súa programabilidade e proporcionando un rendemento razoábel.

Nesta disertación consideramos tres tipos de problemas cuxa paralelización non se axusta ben ás ferramentas máis utilizadas por diferentes motivos: o patrón divide-e-vencerás, algoritmos con paralelismo de datos amorfo, e as aplicacións baseadas en tarefas con patróns arbitrarios de dependencias. Como resultado da nosa análise creamos unha solución baseada en librarías, adaptada para cada un destes problemas en sistemas de memoria compartida. As nosas librarías foron desenvolvidas en C++, xa que é unha linguaxe moi popular que prove tanto alto rendemento como excelentes ferramentas para expresar abstraccións de alto nivel. O framework subxacente empregado polas nosas propostas para crear e xestionar o paralelismo é a libraría Intel Threading Building Blocks (TBB) [112], xa que está dispoñíbel amplamente e mostrou un comportamento mellor que o doutras alternativas nos tests que realizamos antes de desenvolver a versión final das nosas librarías. Grazas a isto e ás múltiples optimizacións aplicadas nas nosas propostas, o rendimento que obteñen é competitivo co doutros opcións dispoñíbeis, e ofrecendo melloras de programabilidade na maioría dos experimentos.

Dos tres problemas abordados, o primeiro, que é a paralelización do tradicional patrón divide-e-vencerás, é o máis coñecido. A pesar deste feito, e da enorme relevancia deste patrón, non atopamos un esqueleto flexíbel baseado en abstraccións

de alto nivel para a súa implementación en sistemas de memoria compartida. A necesidade de tal esqueleto foi motivada nesta tese tras o análise dos problemas da súa implementación co esqueleto máis similar provisto pola libraría de esqueletos máis empregada actualmente, as Intel TBB. A nosa proposta, que demos en chamar `parallel_recursion`, usa un obxecto para proporcionar a información da estrutura e da descomposición da entrada do problema e outro para prover as operacións a realizar. O uso do noso esqueleto resultou en códigos entre 2.9 e 4.6 veces máis curto en termos de liñas de código respecto das implementacións con TBB cando só se tiña en conta as porcións dos códigos afectadas pola paralelización. Mesmo considerando a aplicación completa e unha métrica máis precisa como o esforzo de programación, que ten en conta o número e variedade de símbolos usados no código, `parallel_recursion` necesitou un 14.6 % menos de esforzo que as TBB estándar para a paralelización destes algoritmos. Tamén percibimos que no caso específico de algortimos que non necesitan de ningunha función para combinar os resultados dos seus subproblemas para construiren o resultado final e que están baseados en arrays, que se axustan naturalmente aos rangos nos que as plantillas das TBB están baseadas, as TBB obtiveron unhas métricas de programabilidade mellores que as da nosa libraría, necesitando un esforzo un 11.7 % menor. En canto ao rendemento, o noso esqueleto comportouse de media mellor que as implementacións con TBB ou OpenMP nas dúas máquinas probadas, cando se empregaba particionamento automático, aínda que a selección manual da granularidade podería permitir que as TBB tivesen mellor rendemento que `parallel_recursion` nunha das máquinas.

O paralelismo de datos amorfo é o segundo problema, e posibelmente o máis complexo, considerado nesta disertación, dada a natureza altamente irregular e as condicións dinamicamente cambiantes que caracterizan as aplicacións que se axustan a este paradigma. Esta tese propón a paralelización destes problemas estendendo o paralelismo de datos con potentes abstraccións [18] e aplicando o ben coñecido concepto de esqueleto [57] a este novo campo. Deste xeito, a nosa proposta é un esqueleto chamado `parallel_domain_proc` que se basea na abstracción dun dominio no que os elementos a procesar son definidos. O noso esqueleto usa este dominio tanto para particionar o traballo, mediante a subdivisión recursiva do dominio de entrada, como para detectar conflitos potenciais entre computacións paralelas, comprobando a propiedade dos elementeos a acceder polo subdominio a considerar. O esqueleto é completamente agnóstico con respecto ao obxecto que representa a estructura

irregular a procesar, e unicamente require que poida soportar actualizacións concurrentes desde tarefas paralelas, e ten uns requerimentos limitados sobre a API e a semántica dos obxectos do dominio. Adicionalmente, o feito de que as comprobacións do esqueleto para detectar conflitos estean baseadas en condicións calculadas nos elementos a procesar, nomeadamente na súa pertenza ao dominio, máis que nas estratexias baseadas en bloqueos habituais, evita as esperas activas e problemas de contención que usualmente se asocian aos bloqueos. Outra vantaxe do noso enfoque é que os elementos de traballo se examinan como moito unha vez por nivel de subdivision do dominio de entrada, o que define un límite claro no máximo número de intentos para procesalos. Nos nosos experimentos, as versión paralelas desenvolvidas usando o noso esqueleto necesitaron como máximo un 3 % de liñas de código adicionais con respecto ás versións secuenciais, mentres que empregaron de feito aínda menos sentencias condicionais no código cliente, o que se reflicte nun número ciclomático máis pequeno, grazas á inclusión na nosa libraría de varios dos bucles e comprobacións necesarias. Respecto ao rendemento, mostramos que nestas aplicacións depende en grande medida de diversos factores que o noso esqueleto permite axustar, tales como a política de descomposición de traballo, a granularidade das tarefas o as estruturas de datos utilizadas. Finalmente, unha comparación cualitativa co traballo relacionado indica que as aceleracións obtidas coa nosa libraría estan á par das obtidas empregando outras alternativas, sendo algunhas delas implementacións paralelas manuais.

O terceiro problema que consideramos é a habilidade de expresar do xeito máis conveniente tarefas que deben seguir dependencias de datos arbritarias de modo que ditas dependencias sexan automáticamente impostas. A nosa solución, chamada DepSpawn, necesita que estas tarefas sexan escritas como funcións, que poden ser funcións de C++ normais, mais tamén as convenientes funcións lambda de C++11 ou obxectos `std` :: `function`, de forma que as súas entradas e saídas estean provistas soamente nas súas listas de parámetros. Estas funcións deben ser lanzadas a execución utilizando a funcion `spawn` proporcionada, seguida da súa lista de argumentos. Esta descubre efectivamente cales son as entradas e saídas da función e toma os pasos necesarios para asegurar que a función só se executa cando todas as súas dependencias son satisfeitas. A semántica concreta implementada pola nosa libraría foi coidadosamente descrita, e un tipo de dato especial que ofrece soporte para o procesamento en paralelo de porcións de arrays foi proporcionado, xunto cunha serie de

facilidades para a sincornización explícita. A nosa avaliación revela que as aplicacións baseadas en DepSpawn típicamente conseguen un rendemente igual ou mellor que os códigos desenvolvidos usando OpenMP, xa que pode executar tarefas no momento mesmo no que as súas dependencias propias son satisfeitas e grazas ás vantaxes das TBB con respecto a OpenMP. Igual que nos outros problemas considerados, a nosa solución normalmente resultou nunhas métricas de programabilidade mellores que as dos códigos paralelizados con OpenMP. Adicionalmente, unha discusión detallada que examinou alternativas existentes tanto funcionais como imperativas mostrou que DepSpawn é ou ben máis xeral ou presenta varias vantaxes de programabilidade e rendemento con respecto ás propostas previas. No caso das alternativas imperativas máis cercanamente relacionadas a razón é que ou ben están orientadas a campos específicos de aplicación ou necesitan máis información das usuarias e presentan máis restricións na súa aplicabilidade.

Principais contribucións

- Análise das ferramentas dispoñíbeis para a programación paralela en sistemas de memoria compartida.
- Deseño e desenvolvemento de solucións para problemas que non se adaptan ás ferramentas existentes:
  - Patrón de paralelismo divide-e-vencerás.
  - Algoritmos con paralelismo de datos amorfo.
  - Programas con patróns arbitrarios de dependencias entre tarefas superescalares.
- Implementación destas solucións en forma de librarías de aplicabilidade xenérica e fácil programabilidade.
- Estudo do rendemento e da programabilidade obtidos con estas librarías.

Publications from the thesis

- Carlos H. González, Basilio B. Fraguela, Enhancing and Evaluating the Configuration Capability of a Skeleton for Irregular Computations, 23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing, accepted for publication, 2015
- Carlos H. González, Basilio B. Fraguela, An Algorithm Template for Domain-Based Parallel Irregular Algorithms, International Journal of Parallel Programming, 42 (6), pp. 948–967, 2014
- Carlos H. González, Basilio B. Fraguela, A framework for argument-based task synchronization with automatic detection of dependencies, Parallel Computing, 39 (9), pp. 445–489, 2013
- Carlos H. González, Basilio B. Fraguela, An Algorithm Template for Domain-Based Parallel Irregular Algorithms Proceedings of the International Symposium on High-level Parallel Programming and Applications (HLPP2013), 2013
- Carlos H. González, Basilio B. Fraguela, A framework for argument-based task synchronization, Proceedings of the 16th Workshop on Compilers for Parallel Computing (CPC'12), 2012
- Carlos H. González, Basilio B. Fraguela, An algorithm template for parallel irregular algorithms, Proceedings of the ACACES 2011, pp. 163–166., 2011
- Carlos H. González, Basilio B. Fraguela, A generic algorithm template for divide-and-conquer in multicore systems, Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications 2010 (HPCC'10), pp. 79–88, 2010

- Carlos H. González, Basilio B. Fraguela, Una plantilla genérica para el patrón de paralelismo divide-y-vencerás en sistemas multinúcleo, Actas das XXI Jornadas de Paralelismo (JP'10), pp. 19–26, 2010

Contents

1. Introduction **1**

1.1. Parallelism classification . . . 3

1.1.1. Process interaction . . . 3

1.1.2. Problem decomposition . . . 4

1.1.3. Data structures . . . 5

1.2. The problem . . . 6

1.3. Thesis approach and contributions . . . 10

1.4. Language and tools . . . 11

1.4.1. Threading Building Blocks . . . 12

1.5. Programmability metrics . . . 15

2. Parallel skeleton for divide-and-conquer algorithms **19**

2.1. Divide-and-conquer with the TBB . . . 20

2.1.1. Fibonacci numbers . . . 20

2.1.2. Tree reduction . . . 23

2.1.3. Traveling salesman problem . . . 25

2.2. An algorithm template for divide-and-conquer problems . . . 29

2.2.1. Examples of usage . . . 32
2.3. Evaluation . . . 34
2.3.1. Programmability . . . 36
2.3.2. Performance . . . 37
2.4. Related work . . . 40
2.5. Conclusions . . . 41

3. Parallel skeleton for domain-based irregular algorithms 43
3.1. Domain-Based Parallel Irregular Algorithms . . . 44
3.2. A parallelization scheme based on domains . . . 46
3.2.1. Recursive subdivision . . . 47
3.2.2. Conflict detection . . . 49
3.2.3. Generation of new workitems . . . 50
3.2.4. Domain merging . . . 50
3.2.5. Discussion . . . 51
3.3. The library . . . 52
3.3.1. Graph . . . 52
3.3.2. Worklist . . . 53
3.3.3. Domain . . . 54
3.3.4. Operation . . . 55
3.3.5. parallel_domain_proc skeleton . . . 56
3.4. Tested algorithms . . . 58
3.4.1. Boruvka . . . 58
3.4.2. Delaunay mesh refinement . . . 59

3.4.3. Graph labeling . . . 61
3.4.4. Spanning tree . . . 62
3.5. Evaluation . . . 63
3.6. Exploring the Configuration Capabilities of the Skeleton . . . 70
3.6.1. Skeleton Behavior Configuration . . . 71
3.6.2. Evaluation . . . 74
3.7. Related work . . . 80
3.8. Conclusions . . . 84

4. Library for task parallelism with detection of dependencies 85
4.1. DepSpawn: An argument-based synchronization approach . . . 86
4.1.1. Spawning parallel tasks . . . 87
4.1.2. Array support . . . 92
4.1.3. Explicit synchronization facilities . . . 94
4.1.4. Implementation details . . . 95
4.2. Tested algorithms . . . 97
4.2.1. N-body simulation using Barnes-Hut . . . 97
4.2.2. LU decomposition . . . 98
4.2.3. Cholesky decomposition . . . 99
4.2.4. Sylvester equations resolution . . . 100
4.3. Evaluation . . . 101
4.4. Related work . . . 108
4.5. Conclusions . . . 113

5. Conclusions 115

5.1. Future Work 118

References **121**

List of Listings

2.1. Computation of the n-th Fibonacci number using TBB's `parallel_reduce` 22
2.2. Reduction on a 3-ary tree using TBB's `parallel_reduce` (a) 24
2.3. Reduction on a 3-ary tree using TBB's `parallel_reduce` (b) 25
2.4. Range and body for the Olden tsp parallelization using TBB's `parallel_reduce` (a) 27
2.5. Range and body for the Olden tsp parallelization using TBB's `parallel_reduce` (b) 28
2.6. Templates that provide the pseudo-signatures for the info and body objects used by `parallel_recursion` 30
2.7. Pseudocode of the `parallel_recursion` algorithm template 31
2.8. Computation of the n-th Fibonacci number using `parallel_recursion` 33
2.9. Reduction on a 3-ary tree using `parallel_recursion` 33
2.10. Olden tsp parallelization using `parallel_recursion` 34
3.1. Common pseudocode for an algorithm that uses irregular data structures 44
3.2. Required interface for the `Worklist` class 54
3.3. Required interface for the `Domain` class 54
3.4. Pseudocode of the Boruvka minimum spanning tree algorithm 59
3.5. Pseudocode of the Delaunay mesh refinement algorithm 60
3.6. Pseudocode of the graph labeling algorithm 61
3.7. Pseudocode of the spanning tree algorithm 62
3.8. Serial version of Boruvka's algorithm 63
3.9. Parallel version of Boruvka's algorithm 64
4.1. Enforcing dependencies between tasks (a) Wrong 91
4.2. Enforcing dependencies between tasks (b) Right 92

4.3. Example of definition of an array and a subarray. 93
4.4. Usage of the `Array` class to enable the parallel processing of independent tasks. 94
4.5. Pseudocode of the parallel implemenation using `spawn` of the Barnes-Hut algorithm 97
4.6. Pseudocode of the LU decomposition 100
4.7. Pseudocode of the Cholesky decomposition 101
4.8. Pseudocode of the Sylvester equations solver 102

List of Figures

2.1. Productivity statistics with respect to the OpenMP baseline version of TBB based (TBB) and `parallel_recursion` based (pr) implementations. SLOC stands for source lines of code, eff for the programming effort and cn for the cyclomatic number. 36

2.2. Performance of fib 38

2.3. Performance of merge 38

2.4. Performance of quicksort 38

2.5. Performance of nqueens 38

2.6. Performance of treeadd 39

2.7. Performance of bisort 39

2.8. Performance of health 39

2.9. Performance of tsp 39

3.1. Structure of the domain-based parallelization of irregular algorithms . 48

3.2. Example of an edge contraction of the Boruvka algorithm 58

3.3. Retriangulation of cavities around bad triangles 60

3.4. Relative percentages of the SLOCs and the cyclomatic number of the parallelized version with respect to the sequential one 65

3.5. Running times for the benchmarks 67

3.6. Speedups with respect to optimized serial versions 68

3.7. Relative speedup with respect to no over-decomposition in runs with 8 cores. 100 is the baseline, that is, achieving 100% of the speedup (i.e. the same speedup) obtained without overdecomposition. 69

3.8. Percentage of out-of-domain elements running with 8 cores and 16 bottom-level subdomains 69

3.9. Speedups using different domains and containers 76

3.10. Speedups using different domains and containers. In this figure, lists use our pool allocator 78

3.11. Speedups in the experiments with lists using the pool allocator with respect to the lists using the standard allocator in the runs with 16 cores 79

3.12. Speedups using different levels of decomposition with respect to no over-decomposition in runs with 16 cores 81

4.1. Preceding tasks $P_{T_{42}} = \{T_1, T_2, T_3, T_{41}\}$ for task T_{42}. Assuming that out of them only T_1 generates dependencies with T_{42} (i.e. $DP_{T_{42}} = \{T_1\}$), Descendants$(DP_{T_{42}}) = \{T_{11}, T_{12}, T_{13}, T_{131}, T_{132}\}$ is also depicted. 90

4.2. Blocked algorithm for computing the LU decomposition of a square matrix 99

4.3. Dependencies of the Sylvester equation solver. Nodes refer to calls in Listing 4.8 103

4.4. N-body simulation with Barnes-Hut algorithm in the i7 system 104

4.5. N-body simulation with Barnes-Hut algorithm in the Xeon system . . 104

4.6. LU decomposition in the i7 system 105

4.7. LU decomposition in the Xeon system 105

4.8. Cholesky decomposition in the i7 system 106

4.9. Cholesky decomposition in the Xeon system 106

4.10. Sylvester equations in the i7 system 107

4.11. Sylvester equations in the Xeon system 107

4.12. Small DepSpawn code and associated dataflow graph. 111

List of Tables

2.1. Benchmarks used 35

3.1. Baseline times for the algorithms 75

4.1. Behavior of `spawn` for each parameter type. `A` is any arbitrary data type. Modifiers in brackets do not change `spawn`'s behavior. 87

4.2. Programmability metrics 108

Chapter 1

Introduction

In the past decades, the performance of computer processors has been steadily increasing thanks mainly to the improvement of fabrication processes and the advances in microarchitecture. Smaller transistors mean that more functional units can be packed in the same silicon area; higher clock frequencies make for faster processing. But the current fabrication technology has an upper limit where no further developments can be made, because transistors can not be made to switch faster. Thus, since the 2000s, there has been a rebirth of the research on multiprocessors, and multicore processors, where several cores coexist in the same chip [68]. But as opposed to increasing frequencies or microarchitectural improvements, where a new processor almost automatically allows faster programs without needing changes, multicore or multiprocessor systems require different programming models and tools to achieve better results.

The most widely used tools for parallel programming available nowadays fall into three categories. This way, many tools provide high level abstractions and notations that are well suited for efficiently expressing and thus exploiting the hardware available for regular and data parallel algorithms. Unfortunately, the higher the complexity of the data structures used and the patterns of dependencies among the parallel tasks in which an algorithm can be decomposed, the more unfit these tools are to parallelize such algorithm. A second category are transactional approaches [69], heavily studied in the past years, which allow to parallelize irregular problems with a simple high-level notation, but they either require specific hardware,

or they can incur in large performance overheads[23]. As a result, when considering applications with irregularities and complex patterns of parallelism users often have in practice to resort to the third widespread kind of tools to parallelize codes, which provide a low level of abstraction. These tools often only consist of communication and synchronization primitives, leading to steep learning curves and heavily reduced programmer productivity.

The popularization of multicore systems, even for standard personal computers, calls for easier and more general parallel programming tools, so that programmers can easily take advantage of this computer power for every kind of applications. However, most of the research on parallel computing has focused on scientific problems based on regular structures and simple patterns of parallelism in which computations are relatively easy to distribute among different cores and dependencies are known in advance. As a result, the vast majority of the currently available tools to facilitate parallel programming using a high level of abstraction are oriented to this kind of algorithms. This way more work is needed to assist the parallelization of algorithms that exhibit non trivial patterns of tasks dependencies and communications, as they are currently ill-supported. For this thesis, we studied a wide sample of algorithms whose efficient parallelization requires non negligible user effort using the existing tools, and found several categories with common characteristics. We described the main defining points of each category, and with this knowledge we developed corresponding library-based solutions that allow an easier implementation of the identified algorithm classes.

Following this introduction, this chapter first describes the different types of parallelism along different dimensions in order to then define the motivation and the scope of the work performed in this thesis and its contributions. Then, Section 1.4 briefly justifies the language chosen for the libraries and benchmarks in this dissertation, and it describes the underlying parallelization library used by our proposals. Finally, this chapter defines three metrics based on the source code that will be used during the thesis to measure the programmability of the different approaches tried.

1.1. Parallelism classification

When implementing an algorithm, usually there exist several ways to make a parallel implementation. A careful analysis of the algorithm, its data structures and the available hardware helps to find which solution is better suited. The possible parallel implementations can be classified along several dimensions. For the purposes of this work it suffices to take into account three of them:

1.1.1. Process interaction

There are different ways a system can use when dealing with multiple processes running concurrently. Historically, the hardware for which the program was developed determined the model that should be used [36], and this is still true when developing middleware. Nowadays, however, there exist software tools and hardware support for different combinations of process interaction models.

Shared memory All the processes share the same memory space. This means they can access all the data for a problem withouth the need for explicit communications and without performance loss, or with a small one, if the underlying hardware permits it. This model performs better on multiprocessor or multicore hardware, where the memory is physically shared. The main drawback of this model is that it introduces inconsistency risks. Two or more processes may try to use the same memory address in the wrong order, producing a result that is not equivalent to the sequential semantics of the program. Shared memory programming tools offer synchronization primitives that, carefully used, help solving this issue.

Distributed memory With this model, each process has its own separate memory space. Before the popularization of multiprocessors, the most common high performance computing platform consisted of clusters of computing nodes connected through a network. While this is still true, these clusters have now hybrid architectures where each node has one or more multicore processors. We mention for completeness that there is also a recent trend of including hardware accelerators such as GPUs in some or all the nodes, but these devices

are out of the scope of this PhD thesis, which is centered on the parallelization on the general purpose CPUs. Although some modern developments can almost transparently hide the distributed nature of these systems and offer a single memory space [108], the usual way to do process communication in these systems is through message passing. The distributed memory model can also be used in a multiprocessor system when the parallel tasks are run by processes that share no logical memory space. The messages are managed by the runtime, that sends them through the network or inside a single node as needed.

1.1.2. Problem decomposition

While in the previous section we classified algorithms according to the way the implementation manages the communications among the parallel tasks, we can also classify them based on how the problem is decomposed. Here we distinguish between data and tasks parallelism [9].

Data parallel This model focuses on data distribution. All the processes run the same code over a set of data that is partitioned among them. This way, parallelism is achieved by dividing the work load in blocks, and each process is responsible for one block. This emphasizes the parallelism present in the data itself. For example, when adding two matrices, processor A can add the top halves and processor B the bottom halves, as these two operations are independent from each other. More complex algorithms may require communication among processes.

Task based This model, instead of working with parallelism at the data level, emphasizes processing level parallelism. When a program does several computations or tasks that are independent of each other, each one can be assigned to a different processing element —e.g. thread or processor— so they run in parallel. This model can derive to the data parallel model if we define the tasks as the computation on a block of the input data, and all the tasks run the same code.

In practice, the term *task parallelism* is applied to frameworks that provide

means to define blocks of code meant to be scheduled to run with the threads available in the system. Usually, there should be more of these tasks than threads.

1.1.3. Data structures

The data structures used in an algorithm play a critical role in its possible parallel implementations, as they give place to different natural data decomposition strategies and task synchronization requirements [90]. We describe two main classes of data structures in relation to parallelism, according to the characteristics of their topology [104].

Regular structures The defining characteristic of regular structures is that the pattern of relation between its data components is simple, and usually in terms of their position within the structure, and their topology does not change at runtime.

Simple relation patterns between data items mean that given an element, the program can compute where its successors and predecessors are stored in memory from just its position in the structure. This definition is, thus, implementation dependent. For example, a binary tree can be considered a regular structure if it is stored lineally in a vector, or irregular if it is built with pointers to arbitrarily allocated nodes.

The paradigm of regular structures is a dense vector or matrix, where all the elements are stored consecutively, and each one can be identified with a vector of coordinates.

Irregular structures As opposed to regular structures, these ones are not predictable and their topology can change. Because of this, it is hard or even impossible to know in advance how the elements are stored. Additionaly, given that their topology can change —adding new items with arbitrary relations with the existing ones, or removing some elements, relations, or both—, the execution flow of the program cannot be defined beforehand. Irregular structures can be modeled in general as graphs.

Regular structures usually lead to regular algorithms that fall into the data parallel model. Irregular structures, on the other hand, are better processed with irregular algorithms implememented using a task based framework, as it can better handle the introduction of arbitrary execution units at run time. Also, this kind of algorithms benefit of a more specific analysis, better suited for them than the static dependency analysis used for regular algorithms. Irregular algorithms can be described in terms of a data-centric formulation, called the operator formulation [104]. This formulation is independent of the implementation of the data structure. It defines which are the active elements —those that might need some computation—, their neighborhoods —elements connected to the active ones that are required in the computation—, and the ordering of the computations. In some cases, changing this ordering may produce different but still valid results, which is called do-not-care nondeterminism.

1.2. The problem

During the past years, extensive research has been carried out on the best ways to express parallelism. This has led to an evolution from low level tools [20][51][59] to a variety of new higher level approaches. The large majority of these tools (e.g. [19][24][30][34][45][46][48][70][121]) are well suited to parallelize regular algorithms, whose computations are relatively easy to distribute among different processing elements. The reason is that parallelism has been restricted for a long time to scientific computing systems where regular structures such as matrices and vectors predominate and where most of the critical algorithms allow relatively simple parallelization patterns. However, now that basic physical restrictions such as the power wall have slowed down the increase of performance of individual processors, giving place to the ubiquitous presence of multicore processors in every kind of system, from mobile phones to supercomputers, parallelization is becoming the main way to improve the performance of every kind of applications.

Unfortunately, in many fields irregularity arises in very different forms, usually due to the use of irregular, usually pointer-based, data structures [104]. These applications require a different approach, as it is more complex, and sometimes even impossible to find an a priori distribution of work in them that avoids conflicts

among the parallel threads of execution and balances their workload. Tracking these conflicts is also complicated by the lack of regularity and the potential dynamic changes in the relations among the data items that participate in a computation, synchronization mechanisms being usually required before accessing each element to process.

As a result of this situation the parallelization of irregular algorithms typically requires much more work from programmers, which calls for new tools to deal with irregularity in parallel applications [103]. Solutions to reduce the programming effort of parallel codes can come in the form of new programming languages [22][24][28][97][129], compiler directives [99][105][101][5][49], or libraries [11][96][114][26][16][112][42][95]. New programming languages have many drawbacks because they usually have longer learning curves than the other approaches and they force programmers to rewrite their applications from scratch, an effort that can be ameliorated by providing interfaces with codes and libraries written in other languages. In addition, the need for compiler support adds complexity to their implementation and makes more difficult their extension. Compiler directives suffer from similar problems regarding their implementation [3] and the requirement to use specifically enabled compilers. Also, the fact that they are usually inserted in sequential code, i.e., code that must run successfully in a sequential manner when they are ignored, reduces their ability to provide structure and functionality to applications in comparison with the other approaches [34].

Given the reasons described above, we think that libraries, with their inexistent or small compiler requirements, relatively short learning curves, and their promotion of code reuse, are a promising approach to improve the programmability of parallel applications if they are carefully designed to avoid performance problems [47]. One of the best options to hide the complexity of the parallelization of irregular applications by means of libraries is the use of skeletons [33][57]. Built on parallel design patterns, skeletons provide a clean specification of the flow of execution, parallelism, synchronization and data communications of typical strategies for the parallel resolution of problems. Unfortunately, most skeleton libraries [19][30][34][45][46][121] focus on regular problems. Parallel libraries that can support specific kinds of irregular algorithms exist [17][7], but there are only a few general-purpose developments based on broad abstractions. This dissertation tries thus to help programmers de-

velop parallel applications that exhibit some kind of parallelism using skeletal-like libraries. The hardware scope chosen has been that of current multi-core and/or multiprocessor shared-memory machines. The reasons are that these systems are ubiquitous since the appearance of multi-core processors, and that the flexibility they provide in terms of data sharing and synchronization favors the implementation of irregular applications, which has turned them into the primary target for current research on the parallelization of this kind of algorithms [76][85][117][1][12].

The development of skeletal libraries requires finding recurring patterns whose management can be automated using higher-order functions. During our study of collections of algorithms based on pointer-based data structures [115][77] we found two common patterns that fall into a parallel algorithm design pattern [90] and are amenable to be implemented as skeletons with suitable abstractions. Later, we looked at the wider problem of enabling the efficient expression and execution of parallel computations composed by tasks with arbitrary patterns of dependencies. These three subjects are now commented in turn.

The analysis of well-known pointer-based codes such as the Olden benchmark suite [115] reveals that many of them are based on tree or tree-like data structures. While being irregular data structures, trees can be easily subdivided in subsets that can be processed in parallel, the result of the processing of each one of those subsets being a component of the final result. This way one can proceed from the initial problem, dividing it repeatedly in smaller subproblems until some condition is met. The intention of this subdivision is to find smaller problems that are easier to tackle and enable parallelism by solving them in parallel. This is in fact the divide-and-conquer pattern of parallelism [90]. While this pattern is well established, and there are several skeleton-based implementations [34][30][6][46], they do not provide the degree of generality and minimal programming effort that we were seeking, and most of them target distributed memory systems. For this reason this thesis proposes a new algorithm template for this pattern [52] that provides high performance, great programmability and a totally general scope of application.

The second problem considered in this dissertation is the attempt to provide more abstractions and structure to the exploitation of amorphous data-parallelism [76]. This is an analogue to data parallelism in regular algorithms, but operating on data in irregular data structures such as graphs instead of regular structures such as

matrices, which gives place to more dynamic and changing relations between data and tasks. The main data structure found in algorithms that exhibit this parallelism is usually a graph or something equivalent, where a series of active nodes can be identified. The active nodes are the ones that require some processing at a given time during the execution of the program, and such processing often requires not only accessing them, but also a part of the graph around them called the neighborhood. Amorphous data-parallel codes can dynamically create and remove active nodes as well as elements in their neighborhoods, and the extent of those neighborhoods can often only be computed when the active node processing begins. As a result, the complexity of the parallelization of these applications is enormous. For these reasons, and given the already commented lack of skeleton-based approaches to support the parallelization of irregular algorithms despite the attractive properties of skeletons, this thesis sought to make contributions in this area based on the idea of extending data parallel programming with new abstractions [18]. This way, this dissertation presents a skeleton with some supporting abstract data types, which based on the abstraction of domain, enables the efficient parallelization of a large kind of amorphous data-parallel algorithms [55]. We also illustrate the flexibility of this skeleton, which allows to change with little effort the work partitioning strategy and granularity as well as the data structures used, showing the influence of these decisions on the performance [56].

The last problem tackled in this thesis is that of the development of parallel applications composed by tasks that can present arbitrary patterns of dependencies, which can be thus regular or irregular. The preceding problems considered, while operating on irregular data structures, required the user to find an underlying pattern to identify a suitable skeleton and then fill in its application-specific operations. Here, however, we look at any arbitrary program as a set of potentially parallel tasks that must be ordered according to their data dependencies. This view has the potential to allow the parallel execution of any kind of application, no matter it presents irregularity or not in its data structures and/or patterns of dependencies among tasks. It only requires, however, a minimal abstraction from the user, who just needs to identify the inputs and the outputs of each task to run, resulting in a very simple and powerful approach. While many researchers have already considered this possibility [26][101][125][84][88][86], the use of functional languages, with their restrictive characteristics, or the restriction to an specific kind of application and

lack of a general programming model, or the need to programmatically specify the dependencies of each task of the solutions provided did not satisfy us. This way, the last stage of this PhD Thesis proposes a general library-based solution implementing a programming model that allows expressing and running arbitrary task-parallel applications with minimal programming cost and maximal flexibility [54].

1.3. Thesis approach and contributions

Implementing irregular algorithms, and particularly parallelizing them, is usually harder than regular ones. The data can be located in arbitrary positions in memory, with no clear way to predict where. Also, the relations between elements and/or parallel tasks may not follow a clearly defined pattern. Because of these reasons, it is cumbersome, or plainly impossible, to use the same methods available for regular data parallel algorithms. The most practical approach in these situations is usually to develop a task based parallel program applying some low level API because most current parallel programming tools offer no good abstractions for irregular problems.

The purpose of this thesis is to help advance in the parallelization of applications that present any of the complexities described in the preceding Section. The approach taken is to define simple high level abstractions that facilitate the parallelization of applications that present one of the difficulties tackled and to implement libraries that allow to express and exploit such ideas. Concretely, this thesis presents three libraries, each one of them suited to parallelize a different kind of algorithm:

parallel_recursion Provides a skeleton that implements the divide-and-conquer pattern after analyzing and providing abstractions for each one of its components. The user must provide two objects besides the input data. One of them describes the operations to perform during the solution of the problem, such as the resolution of the base case and the combination of the partial results. The other object describes the problem by providing methods to identify base cases as well as to obtain the number of children and to subdivide non base cases.

parallel_domain_proc This module provides an purely skeleton-based approach

to parallelize applications that present amorphous data-parallelism. The input of the skeleton is a graph where the nodes have attributes that can be described as belonging to some domain. For example, in a graph of triangles, each node has coordinates on a plane; this plane would be the domain. The programmer then provides a definition of the domain, that the skeleton can use to logically partition the graph, generating parallel tasks that operate on disjoint subdomains, and to detect conflicts. The parallelization of this kind of applications under our scheme allows for a large number of variations for the work decomposition, data structures, etc. Our library also provides tools to explore many of these possibilities, as our experiments show that they have an enormous impact on performance.

dependent spawn (DepSpawn) This is a system that allows to easily request the execution of parallel tasks while automatically detecting the dependencies among such tasks. Our library supports tasks defined as regular C++ callable objects —e.g. functions, lambdas or functors—. By means of an analysis of the parameters passed to the callable object, the library can build a dependency graph for the tasks, and use it to order their execution.

1.4. Language and tools

A crucial decision in any software project is choosing the implementation language. While there are more widespread languages such as C, C++ was chosen for this thesis. The most important reason for this are the capabilities of object-oriented languages such as C++ for the development of libraries. These languages offer mechanisms that make it easier to attain critical objectives of libraries such as providing an adequate representation of the semantics of the problems the programmer wants to implement and hiding the particularities of the runtime. Among these languages C++ is particularly well suited for parallel and high performance applications, which are the target of this thesis, as its performance is similar to that of C.

Another advantage is that C++ is easily interoperable with C, so the vast set of libraries available for that language can be used with little effort. This way,

applications written with widely used libraries, such as BLAS, which are common for scientific applications, can benefit of the easy parallelization that the solutions developed in this thesis provide, without needing to change the core of the code.

Once we opted for C++, we exploited its advantages to the fullest, including core language functionality improvements provided by the new C++11 standard. For example, it provides advanced capabilities such as metaprogramming, through the template system, which allows to move part of the computations to compile time, as well as to enable type analysis and reflection, which is fundamental especially for the task dependency library presented in Chapter 4. The C++11 standard introduces variadic templates, which allow to use an arbitrary number of parameters for templates. The skeletons developed in this thesis also provide support for lambda functions, so that the operations can be expressed with a clear syntax that greatly resembles that of the standard language constructs, for example `for` loops.

1.4.1. Threading Building Blocks

Intel Threading Building Blocks (TBB) [112] is a C++ library developed by Intel for the programming of multithreaded applications. This library is available on the FreeBSD, Linux, Solaris, Mac OS X, and Microsoft Windows operating systems and run on top of the x86/x64 (both from Intel and AMD), IA64 (Itanium family) and MIC (new Intel Xeon Phi) processors. There are also ports for the Xbox 360 and PowerPC-based systems. This way, TBBs support the vast majority of current computers. It provides from atomic operations and mutexes to containers specially designed for parallel operation. Still, its main mechanism to express parallelism are algorithm templates that provide generic parallel algorithms. The most important TBB algorithm templates are `parallel_for` and `parallel_reduce`, which express element-by-element independent computations and a parallel reduction, respectively. These algorithm templates have two compulsory parameters. The first one is a *range* that defines a problem that can be recursively subdivided into smaller subproblems that can be solved in parallel. The second one, called *body*, provides the computation to perform on the range. The requirements of the classes of these two objects are now discussed briefly.

The ranges used in the algorithm templates provided by the TBB must model

the Range concept, which represents a recursively divisible set of values. The class must provide:

- a copy constructor
- an `empty` method to indicate when a range is empty,
- an `is_divisible` method to inform whether the range can be partitioned into two subranges whose processing in parallel is more efficient than the sequential processing of the whole range,
- a splitting constructor that splits a range `r` in two. By convention this constructor builds the second part of the range, and updates `r` (which is an input by reference) to be the first half. Both halves should be as similar as possible in size in order to attain the best performance.

TBB algorithm templates use these methods to partition recursively the initial range into smaller subranges that are processed in parallel. This process, which is transparent to the user, seeks to generate enough tasks of an adequate size to parallelize optimally the computation on the initial range. Thus, TBB makes extensive usage of a divide-and-conquer approach to achieve parallelism with its templates. This recursive decomposition is complemented by a task-stealing scheduling that balances the load among the existing threads, generating and moving subtasks among them as needed.

The body class has different requirements depending on the algorithm template. This way, `parallel_for` only requires that it has a copy constructor and overloads the `operator`() method on the range class used. The parallel computation is performed in this method. `parallel_reduce` requires additionally a splitting constructor and a `join` method. The splitting constructor is used to build copies of the body object for the different threads that participate in the reduction. The `join` method has as input a `rhs` body that contains the reduction of a subrange just to the right of (i.e following) the subrange reduced in the current body. The method must update the object on which it is invoked to represent the accumulated result for its reduction and the one in `rhs`, that is, left.`join`(right) should update left to be the result of left reduced with right. The reduction operation should be associative,

but it need not be commutative. It is important that a new body is created only if a range is split, but the converse is not true. This means that a range can be subdivided in several smaller subranges which are all reduced by the same body. When this happens, the body always evaluates the subranges in left to right order, so that non commutative operations are not endangered.

TBB algorithm templates have a third optional parameter, called the *partitioner*, which indicates the policy followed to generate new parallel tasks. When not provided, it defaults to the `simple_partitioner`, which recursively splits the ranges giving place to new subtasks until their `is_divisible` method returns `false`. Thus, with it the programmer fully controls the generation of parallel tasks. The `auto_partitioner` lets the TBB library decide whether the ranges must be split to balance load. The library can decide not to split a range even if it is divisible because its division is not needed to balance load. Finally, `affinity_partitioner` applies to algorithms that are performed several times on the same data and these data fit in the caches. It tries to assign the same iterations of loops to the same threads that run them in a past execution.

Our libraries use the low level API of the TBBs, which is based in the task parallel model described before. One of the main benefits of this approach is that our libraries enjoy the smart load balancing and task affinity strategies provided by TBBs. For example, instead of a global task queue that could be a source of contention, each thread has its local pool of tasks. This besides promotes locality because the tasks spawned by a given thread are more likely to be run by this thread, which increases the potential to reuse data accessed by the ancestor tasks in the caches of the same core. TBBs also provide a clever task stealing mechanism that allows to perform load balancing between threads in a completely transparent way. Idle threads steal the least recently generated tasks of busy threads, which are the tasks with less locality and potentially more work. Finally, TBBs also help deal with the memory model of the underlying hardware under the control of its API, which provides accesses with the release and acquire semantics as well as sequentially consistent accesses that act as memory fences. Our libraries use this API to guarantee the correct execution both of its internal code as well as the user functions under any memory model.

1.5. Programmability metrics

Since the final purpose of this thesis is to facilitate the work of programmers when developing parallel applications, measuring the impact of the usage of our libraries on programmability is critical to evaluate our success. Unfortunately, programmability is hard to measure, and while it would be ideal to base our analysis on the measurement of the devolpment times, quality of the code written, opinions, etc. of teams of programmers [122] such teams are seldom available. For this reason, many studies on programmability rely on objective metrics extracted from the code, and this will be the approach followed in this document. Three different metrics will be considered that we now explain in turn.

The most widely known code-based productivity metric is the number of Source Lines of Code (SLOCs), which counts all the lines in the program excluding the comments and the empty lines.

The second metric is the programming effort [63], which goes beyond SLOCs by taking into account the number and variety of tokens (identifiers, keywords, punctuation characters, etc.) used in the program. This way, when computing the programming effort, a program is considered a string of tokens, which can be divided into two groups: *operators* and *operands*. The *operators* are defined as variables or constants that the implementation employs. The *operators* are symbols or combinations of symbols that affect the value or ordering of operands. We denote η_1, the different number of operators; η_2, the different number of operands; N_1, the total number of occurrences of operators; and N_2, the total number of occurrences of operands. The program volume V is defined as

$$V = (N_1 + N_2) log_2(\eta_1 + \eta_2) \tag{1.1}$$

The potential volume V^* describes the shortest possible or most succinct form of an algorithm.

$$V^* = (N_1^* + N_2^*) log_2(\eta_1^* + \eta_2^*) \tag{1.2}$$

Now in the minimal form, neither operators nor operands could require repetition, thus

$$V^* = (\eta_1^* + \eta_2^*) log_2(\eta_1^* + \eta_2^*) \tag{1.3}$$

Furthermore, the minimum possible number of operators η^* for any algorithm is known. It must consist of one distinct operator for the name of the function and another to serve as an assignment or grouping symbol. Therefore,

$$\eta_1^* = 2$$

Equation 1.3 then becomes

$$V^* = (2 + \eta_2^*) log_2(2 + \eta_2^*) \tag{1.4}$$

where η_2^* should represent the number of different input/output parameters. Program level Lvl is defined as

$$Lvl = \frac{V^*}{V} \tag{1.5}$$

It follows that only the most succinct expression possible for an algorithm can have a level of unity. Programming effort E required to generate a given program should be

$$E = \frac{V}{Lvl} \tag{1.6}$$

A further implication of the effort equation can be shown by recalling equation 1.5 and substituting in equation 1.6

$$E = \frac{V^2}{V^*} \tag{1.7}$$

Equation 1.7 indicates that the mental effort required to implement any algorithm with a given potential volume should vary with the square of its volume in any language, rather linearly.

The third metric, the cyclomatic number [91], is a function of the number of predicates in the program. In particular, if a control flow graph (CFG) represents a program containing P decision points or predicates, then the cyclomatic number C is given by

$$C = P + 1 \tag{1.8}$$

Again, the smaller the C, the less complex the program is. In C++, P amounts to the number of constructions that include a predicate on which the control flow depends. These are the `if`, `for`, `while`, and `case` keywords, the `?:` operator, and the conditional preprocessor macros.

While there are several tools that allow to measure the SLOCs [128], this is not the case for the programming effort and the cyclomatic number. For this reason, during the development of this thesis we had to develop a tool to systematically extract these metrics from C and C++ codes, giving place to C3MS (C++ Code Complexity Measurement System) tool, which is freely available under request and which has already been used in several publications [42][39][123][95] in addition to the papers associated to this thesis.

Chapter 2

Parallel skeleton for divide-and-conquer algorithms

The divide-and-conquer strategy appears in many problems [2]. It is applicable whenever the solution to a problem can be found by dividing it into smaller subproblems, which can be solved separately, and merging somehow the partial results to such subproblems into a global solution for the initial problem. This strategy can be often applied recursively to the subproblems until a base or indivisible one is reached, which is then solved directly. The recursivity of an algorithm sometimes is given by the data structure on which it works, as is the case of algorithms on trees, and very often it is the most natural description of the algorithm. Just to cite a few examples, cache oblivious algorithms [50], many signal-processing algorithms such as discrete Fourier transforms, or the linear algebra algorithms produced by FLAME [15] are usually recursive algorithms that follow a divide-and-conquer strategy. As for parallelism, the independence in the resolution of the subproblems in which a problem has been partitioned leads to concurrency, giving place to the divide-and-conquer pattern of parallelism [90]. Its usefulness is recognized also by its identification as a basic parallel programming skeleton [33].

An analysis of suites of programs that operate on pointer-based data structures such as [115] reveals that an important part of such structures represent trees, whose processing can very often be parallelized using the divide- and-conquer pattern. While this pattern is well known, many of its implementations in the form

of parallel skeletons [34][30][6][46] are focused on distributed memory environments, lack generality and/or require more effort from the programmer to apply them than what should be needed in our opinion. We have chosen the Intel Threading Building Blocks (TBB) library [112], presented in Section 1.4, in order to illustrate the kind of problems that we find in the application of the parallel divide-and-conquer pattern given its position as the most popular and widely adopted library for the parallelization of applications in shared memory environments using skeletons. This way, this chapter motivates our proposal with a discussion of the the weaknesses of TBB algorithm templates to parallelize applications that are naturally fit for the divide-and-conquer pattern of parallelism. This is followed by our reasoned proposal of a new template to express these problems, and its evaluation both in terms of programmability and performance.

2.1. Divide-and-conquer with the TBB

This Section analyzes the programmability of the divide-and-conquer pattern of parallelism using the TBB algorithm templates through a series of examples of increasing complexity. This analysis motivates and leads to the design of the alternative that will be presented in the next Section.

2.1.1. Fibonacci numbers

The simplest program we consider is the recursive computation of the nth Fibonacci number. While this is an inefficient method to compute this value, our interest at this point is on the expressiveness of the library, and this problem is ideal because of its simplicity. The sequential version is

```
1 int fib(int n) {
2   if (n < 2) return n;
3   else return fib(n - 1) + fib(n - 2);
4 }
```

which clearly shows all the basic elements of a divide and conquer algorithm:

- the identification of a base case (when n $<$ 2)
- the resolution of the base case (simply return n)
- the partition in several subproblems otherwise (fib(n - 1) and fib(n - 2))
- the combination of the results of the subproblems (here simply adding their outcomes)

The simplest TBB parallel implementation of this algorithm, shown in Listing 2.1, is based on `parallel_reduce` and it indeed follows a recursive divide-and-conquer approach. The range object required by this template is provided by the `FibRange` class, and it encapsulates which is the Fibonacci number to compute in `n_`. The body object belongs to the `Fib` class and performs the actual computation. These classes implement the methods required by the template, which were explained in Section 1.4. For example the splitting constructors are the constructors that have a second dummy argument of type `split`. The splitting constructor for `FibRange` in Lines 7-9 shows how the computation of the n-th Fibonacci number is split in the computation of the $n-1$ and $n-2$th numbers. Concretely, Line 8 fills in the new range built to represent the $n-2$th number, while the input `FibRange` which is being split, called `other` in the code, represents now the $n-1$th number (Line 9). Notice also that the `operator()` of the body (Line 25) must support any value of the input range, and not just a not divisible one (0 or 1). The reason is that the `auto_partitioner` is being used (Line 36) to avoid generating too many tasks. The default `simple_partitioner` would have generated a new task for every step of the recursion, which would have been very inefficient.

The `Fib` class must have a state for three reasons. First, the same body can be applied to several ranges, so it must accumulate the results of their reductions. Second, bodies must also accumulate the results of the reductions of other bodies through the calls to their `join` method. Third, TBB algorithm templates have no return type, thus body objects must store the results of the reductions. This gives place to the invocation we see in Lines 35-37 of the listing. The topmost `Fib` object must be created before the usage of `parallel_reduce` so that when it finishes the result can be retrieved from it.

Altogether, even when the problem suits well the TBB algorithm templates, we

```
struct FibRange {
  int n_;

  FibRange(int n)
  : n_(n) { }

  FibRange(FibRange& other, split)
  : n_(r.n_ - 2)
  { other.n_ = other.n_ - 1; }

  bool is_divisible() const { return (n_ > 1); }

  bool empty() const { return n_ < 0; };
};

struct Fib {
  int fsum_;

  Fib()
  : fsum_(0) { }

  Fib(Fib& other, split)
  : fsum_(0) { }

  void operator() (FibRange& range) { fsum_ += fib(range.n_); }

  int fib(int n) {
    if (n < 2) return n;
    else return fib(n - 1) + fib(n - 2);
  }

  void join(Fib& rhs) { fsum_ += rhs.fsum_; };
};
...
Fib f();
parallel_reduce(FibRange(n), f, auto_partitioner());
int result = f.fsum_;
```

Listing 2.1: Computation of the n-th Fibonacci number using TBB's `parallel_reduce`

have gone from 4 source lines of code (SLOC) in the sequential version to 26 (empty Lines and comments are are not counted) in the parallel one.

2.1.2. Tree reduction

TBB ranges can only be split in two subranges in each subdivision, while sometimes it would be desirable to divide them in more subranges. For example, the natural representation of a subproblem in an operation on a tree is a range that stores a node. When this range is processed by the body of the algorithm template, the node and its children are processed. In a parallel operation on a 3-ary tree, each one of these ranges would naturally be subdivided in 3 subtasks/ranges, one per direct child. The TBB restriction to two subranges in each partition forces the programmer to build a more complex representation of the problem so that there are range objects that represent a single child node, while others keep two children nodes. As a result, the construction and splitting conditions for both kinds of ranges will be different, implying a more complicated implementation of the methods of the range. Moreover, the `operator()` method of the body will have to be written to deal correctly with both kinds of ranges.

Listings 2.2 and 2.3 exemplify this with the TBB implementation of a reduction on a 3-ary tree. The initial range stores the root of the tree in `r1_`, while `r2_` is set to 0 (Lines 5-6). The splitting constructor operates in a different way depending on whether the range `other` to split has a single node or two (Line 9). If it has a single node, the new range takes its two last children and stores that its parent is `other.r1_`. The input range `other` is then updated to store its first child. When `other` has two nodes, the new range takes the second one and zeroes it from `other`. The `operator()` of the body has to take into account whether the input range has one or two nodes, and also whether a parent node is carried.

This example also points out another two problems of this approach. Although a task that can be subdivided in $N > 2$ subtasks can always be subdivided only in two, those two subproblems may have necessarily a very different granularity. In this example, one of the two children ranges of a 3-ary node is twice larger than the other one. This can lead to a poor load balancing, since the TBB recommends the subdivisions to be as even as possible. This problem can be alleviated by further

```
struct TreeAddRange {
  tree_t * r1_, *r2_;
  tree_t * parent_;

  TreeAddRange(tree_t *root)
  : r1_(root), r2_(0), parent_(0) { }

  TreeAddRange(TreeAddRange& other, split) {
    if(other.r2_ == 0) { //other only has a node
      r1_ = other.r1_->child[1];
      r2_ = other.r1_->child[2];
      parent_ = other.r1_;
      other.r1_ = other.r1_->child[0];
    } else { //other has two nodes
      parent_ = 0;
      r1_ = other.r2_;
      r2_ = 0;
      other.r2_ = 0;
    }
  }

  bool empty() const { return r1_ == 0; }

  bool is_divisible() const { return !empty(); }
};
```

Listing 2.2: Reduction on a 3-ary tree using TBB's `parallel_reduce` (a)

subdividing the ranges and relying on the work-stealing scheduler of the TBB, which can move tasks from loaded processors to idle ones. Still, the TBB does not provide a mechanism to specify which ranges should be subdivided with greater priority, but just a boolean flag that indicates whether a range can be subdivided or not. Moreover, when automatic partitioning is used, the library may not split a range even if it is divisible. For these reasons, allowing to subdivide in N subranges at once improves both the programmability and the potential performance of divide-and-conquer problems.

The last problem is the difficulty to handle pieces of a problem which are not a natural part of the representation of its children subproblems, but which are required

```
struct TreeAddReduce {
  int sum_;

  TreeAddReduce()
  : sum_(0) { }

  TreeAddReduce(TreeAddReduce& other, split)
  : sum_(0) { }

  void operator()(TreeAddRange &range) {
    sum_ += TreeAdd(range.r1_);
    if(range.r2_ != 0)
      sum_ += TreeAdd(range.r2_);
    if (range.parent_ != 0)
      sum_ += range.parent_->val;
  }

  void join(TreeAddReduce& rhs) {
    sum_ += rhs.sum_;
  }
};
...
TreeAddReduce tar;
parallel_reduce(TreeAddRange(root), tar, auto_partitioner());
int r = tar.sum_;
```

Listing 2.3: Reduction on a 3-ary tree using TBB's `parallel_reduce` (b)

in the reduction stage. In this code this is reflected by the clumsy treatment of the inner nodes of the tree, which must be stored in the `parent_` field of the ranges taking care that none is either lost or stored in several ranges. Additionally, the fact that some ranges carry an inner node in this field while others do not complicates the `operator()` of the body.

2.1.3. Traveling salesman problem

The TBB algorithm templates require the reduction operations to be associative. This complicates the implementation of the algorithms in which the solution to a

given problem at any level of decomposition requires merging exactly the solutions to its children subproblems. An algorithm of this kind is the recursive partitioning algorithm for the traveling salesman in [72], an implementation of which is the *tsp* Olden benchmark [115]. The program first builds a binary space partitioning tree with a city in each node. Then the solution is built traversing the tree with the function

```
1 Tree tsp(Tree t, int sz) {
2   if (t->sz <= sz) return conquer(t);
3   Tree leftval = tsp(t->left, sz);
4   Tree rightval = tsp(t->right, sz);
5   return merge(leftval, rightval, t);
6 }
```

which follows a divide-and-conquer strategy. The base case, found when the problem is smaller than a size `sz`, is solved with the function `conquer`. Otherwise the two children can be processed in parallel applying `tsp` recursively. The solution is obtained joining their solutions with the `merge` function, which requires inserting their parent node `t`.

This structure fits well the `parallel_reduce` template in many aspects. Listings 2.4 and 2.5 show the range and body classes used for the parallelization with this algorithm template. The range contains a node, and splitting it returns the two children subtrees. The `is_divisible` method checks whether the subtree is smaller than `sz`, when the recursion stops. The `operator()` of the body applies the original `tsp` function on the node taken from the range.

The problems arise when the application of the `merge` function is considered. First, a stack must be added to the range for two reasons. One is to identify when two ranges are children of the same parent and can thus be merged. This is expressed by function `mergeable` (Lines 22-25). The other reason is that this parent is actually required by `merge`.

Reductions take place in two places. First, a body `operator()` can be applied to several consecutive ranges in left to right order, and must reduce their results. This way, when `tsp` is applied to the node in the input range (Line 36), the result is stored again in this range and an attempt to merge it with the results of ranges previously

```
1  struct TSPRange {
2    static int sz_;
3    stack<Tree> ancestry_;
4    Tree t_;
5
6    TSPRange(Tree t, int sz)
7    : t_(t)
8    { sz_ = sz; }
9
10   TSPRange(TSPRange& other, split)
11   : t_(other.t_->right), ancestry_(other.ancestry_)
12   {
13     ancestry_.push(other.t_);
14     other.ancestry_.push(other.t_);
15     other.t_ = other.t_->left;
16   }
17
18   bool empty() const { return t_ == 0; }
19
20   bool is_divisible() const { return (t_->sz > sz_); }
21
22   bool mergeable(const TSPRange& rhs) const {
23     return !ancestry_.empty() && !rhs.ancestry_.empty() &&
24           (ancestry_.top() == rhs.ancestry_.top());
25   }
26 };
```

Listing 2.4: Range and body for the Olden tsp parallelization using TBB's `parallel_reduce` (a)

processed is done in method `mergeTSPRange` (Lines 40-47). The body keeps a list `lresults_` of ranges already processed with their solution. The method repetitively checks whether the rightmost range in the list can be merged with the input range. In this case, `merge` reduces them into the input range, and the range just merged is removed from the list. In the end, the input range is added at the right end of the list. Reductions also take place when different bodies are accumulated in a single one through their `join` method. Namely, left.`join`(right) accumulates in the left body its results with those of the right body received as argument. This can be achieved applying `mergeTSPRange` to the ranges in the list of results of the rhs body

```
1  struct TSPBody {
2    list<TSPRange> lresults_;
3
4    TSPBody() { }
5
6    TSPBody(TSPBody& other, split) { }
7
8    void operator() (TSPRange& range) {
9      range.t_ = tsp(range.t_, range.sz_);
10     mergeTSPRange(range);
11   }
12
13   void mergeTSPRange(TSPRange& range) {
14     while (!lresults_.empty() && lresults_.back().mergeable(range)) {
15       range.t_ = merge(lresults_.back().t_, range.t_, range.ancestry_.top());
16       range.ancestry_.pop();
17       lresults_.pop_back();
18     }
19     lresults_.push_back(range);
20   }
21
22   void join(TSPBody& rhs) {
23     list<TSPRange>::iterator itend = rhs.lresults_.end();
24     for(list<TSPRange>::iterator it = rhs.lresults_.begin(); it != itend; ++it){
25       mergeTSPRange(*it);
26     }
27   }
28 };
29 ...
30 parallel_reduce(TSPRange(root, sz), TSPBody(), auto_patitioner());
```

Listing 2.5: Range and body for the Olden tsp parallelization using TBB's `parallel_reduce` (b)

from left to right (Lines 49-54).

2.2. An algorithm template for divide-and-conquer problems

The preceding Section has illustrated the limitations of TBBs to express divide-and-conquer problems. Not surprisingly, the restriction to binary subdivisions or the lack of associativity impact negatively on programmability. But even problems that seem to fit well the TBB paradigm such as the recursive computation of the Fibonacci numbers have a large parallelization overhead, as several kinds of constructors are required, reductions can take place in several places, bodies must keep a state to perform those reductions, etc.

The components of a divide-and-conquer algorithm are the identification of the base case, its resolution, the partition in subproblems of a non-base problem, and the combination of the results of the subproblems. Thus we should try to enable to express these problems using just one method for each one of these components. In order to increase the flexibility, the partition of a non-base problem could be split in two subtasks: calculating the number of children, so that it need not be fixed, and building these children. These tasks could be performed in a method with two outputs, but we feel it is cleaner to use two separate methods for them.

The subtasks identified in the implementation of a divide-and-conquer algorithm can be grouped in two sets, giving place to two classes. The decision on whether a problem is the base case, the calculation of the number of subproblems of non-base problems, and the splitting of a problem depend only on the input problem. They conform thus an object with a role similar to the range in the TBB algorithm templates. We will call this object the *info* object because it provides information on the problem. Contrary to the TBB ranges, we choose not to encapsulate the problem data inside the info object. This reduces the programmer burden by avoiding the need to write a constructor for this object for most problems.

The processing of the base case and the combination of the solutions of the subproblems of a given problem are responsibility of a second object analogous to the body of the TBB algorithm templates, thus we will call it also body. Many divide-and-conquer algorithms process an input problem of type `T` to get a solution of type `S`, so the body must support the data types for both concepts, although of

```
template<typename T, int N>
struct Info : Arity<N> {
  bool is_base(const T& t) const;     //is t the base case of the recursion?
  int num_children(const T& t) const; //number of subproblems of t
  T child(int i, const T& t) const;   //get i-th subproblem of t
};

template<typename T, typename S>
struct Body : EmptyBody<T, S> {
  void pre(T& t);     //preprocessing of t before partition
  S base(T& t);       //solve base case
  S post(T& t, S *r); //combine children solutions
};
```

Listing 2.6: Templates that provide the pseudo-signatures for the info and body objects used by `parallel_recursion`

course `S` and `T` could be the same. We have found that in some cases it is useful to perform some processing on the input before checking its divisibility and the corresponding base case computation or recursion. Thus the body of our algorithm template requires a method `pre`, which can be empty, which is applied to the input problem before any check on it is performed. As for the method that combines the solutions of the subproblems, which we will call `post`, its inputs will be an object of type `T`, defining the problem at a point of the recursion, and a pointer to a vector with the solutions to its subproblems, so that a variable number of children subproblems is easily supported. The reason for requiring the input problem is that, as we have seen in Sections 2.1.2 and 2.1.3, in many cases it has data which are not found in any of its children and which are required to compute the solution.

Listing 2.6 shows templates that describe the info and body objects required by the algorithm template we propose. The info class must be derived from class `Arity` $<$ `N` $>$, where `N` is either the number of children of each non base subproblem, when this value is a constant, or the identifier `UNKNOWN` if there is not a fixed number of subproblems in the partitions. This class provides a method `num_children` if `N` is a constant. As for the body, it can be optionally derived from the class `EmptyBody` $<$ `T`, `S` $>$, which provides shell (empty) methods for all the methods a body requires. Thus inheriting from it can avoid writing unneeded methods.

```
template<typename T, typename S, typename I, typename B, typename P>
S parallel_recursion(T& t, I& i, B& b, P& p) {
  b.pre(t);
  if(i.is_base(t)) return b.base(t);
  else {
    const int n = i.num_children(t);
    S result[n];
    if(p.do_parallel(i, t))
      parallel_for(int j = 0; j < n ; j++)
        result[j] = parallel_recursion(i.child(j, t), i, b, p)
    else
      for(int j = 0; j < n ; j++)
        result[j] = parallel_recursion(i.child(j, t), i, b, p);
    }
    return b.post(t, result);
  }
}
```

Listing 2.7: Pseudocode of the `parallel_recursion` algorithm template

Listing 2.7 shows the pseudocode for the operation of the algorithm template we propose, which is called `parallel_recursion` for similarity with the names of the standard TBB algorithm templates. Its arguments are the representation of the input problem, the info object, the body object, and optionally a partitioner that defines the policy to spawn parallel subtasks. The figure illustrates the usage of all the methods in the info and body classes, `I` and `B` in the figure, respectively. Contrary to the TBB algorithm templates, ours returns a value which has type `S`, the type of the solution. A specialization of the template allows a return type `void`.

From the pseudocode we see that `Info` :: `is_base` is not the exact opposite of the `is_divisible` method of the TBB ranges. TBB uses `is_divisible` to express divisibility of the range, but also whether it is more efficient to split the range and process the subranges in parallel than to process the range sequentially. Even if the user writes `is_divisible` to return true for all non base cases, the library can ignore it and stop partitioning even if it indicates divisibility if the `auto_partitioner` is used. For these reasons, the `operator`() of a standard body should be able to process both base and non base instances of the range. This makes it different from the `Body` :: `base` method in Listing 2.6, which processes the problem if and only if

`Info` :: `is_base` is true, as line 4 in Listing 2.7 shows.

The decision on whether the processing of the children subproblems is made sequentially or in parallel is up to the partitioner in `parallel_recursion` (lines 8-14 in Listing 2.7). The behavior of the partitioners is as follows. The `simple_partitioner` generates a parallel subtask for each child generated in every level of the recursion, very much as it does in the standard TBB templates. This is the default partitioner. The `auto_partitioner` works slightly different from the standard templates. In them this partitioner can stop splitting the range, even if `is_divisible` is true, in order to balance optimally the load. In `parallel_recursion` this partitioner also seeks to balance load automatically. Nevertheless, it does not stop the recursion in the subdivision of the problem, but just the parallelization in the processing of the subtasks. This way the problem is split always that `Info` :: `is_base` is false. Finally, we provide a new partitioner called `custom_partitioner` which takes its decision on parallelization based on an optional `Info` :: `do_parallel`(`const T&` `t`) method supplied by the user. If this method returns `true`, the children of `t` are processed in parallel, otherwise they are processed sequentially.

Let us now review the implementation of the examples discussed in Section 2.1 using this proposal.

2.2.1. Examples of usage

Listing 2.8 shows the code to compute the n-th Fibonacci number using our `parallel_recursion` skeleton. Compared to the 26 SLOC of the implementation based on `parallel_reduce` in Listing 2.1, this implementation only has 9. This code has the virtue that it not only parallelizes the computation, it even makes unnecessary the original sequential `fib` function thanks to the power of `parallel_recursion` to fully express problems that are solved recursively.

The addition of the values in the nodes of a 3-ary tree, which required 41 SLOC in Listings 2.2 and 2.3, is expressed using 9 SLOC with `parallel_recursion` in Listing 2.9. In fact, the version in Listing 2.2 is a bit longer because it uses the sequential function `TreeAdd`, not shown, to perform the reduction of a subtree in the `operator`() of the body. This function is not needed by the implementation

```
struct FibInfo : public Arity<2> {
  bool is_base(const int i) const { return i <= 1; }

  int child(const int i, const int c) const { return c - i - 1; }
};

struct Fib: public EmptyBody<int, int> {
  int base(int i) { return i; }

  int post(int i, int * r) { return r[0] + r[1]; }
};
...
int result = parallel_recursion<int> (n, FibInfo(), Fib(), auto_partitioner());
```

Listing 2.8: Computation of the n-th Fibonacci number using `parallel_recursion`

```
struct TreeAddInfo : public Arity<3> {
  bool is_base(const tree_t *t) const { return t == 0; }

  tree_t *child(int i, const tree_t *t) const { return t->child[i]; }
};

struct TreeAddBody : public EmptyBody<tree_t *, int> {
  int base(tree_t * t) { return 0; }

  int post(tree_t * t, int *r) { return r[0] + r[1] + r[2] + t->val; }
};
...
int r = parallel_recursion<int>(root, TreeAddInfo(), TreeAddBody(),
                                auto_partitioner());
```

Listing 2.9: Reduction on a 3-ary tree using `parallel_recursion`

based on `parallel_recursion`, which can perform the reduction just using the template.

Our last example, the traveling salesman problem implemented in the *tsp* Olden benchmark, is parallelized with `parallel_recursion` in Listing 2.10. The facts that the `post` method that combines the solutions obtained in each level of the recursion

```
1  struct TSPInfo: public Arity<2> {
2    static int sz_;
3
4    TSPInfo(int sz )
5    { sz_ = sz; }
6
7    bool is_base(const Tree t) const { return (t->sz <= sz_); }
8
9    Tree child(int i, const Tree t) const { return (i == 0) ? t->left : t->right;}
10 };
11
12 struct TSPBody : public EmptyBody<Tree, Tree> {
13   Tree base(Tree t) { return conquer(t); }
14
15   Tree post(Tree t, Tree * results) { return merge(results[0], results[1], t); }
16 };
17 ...
18 parallel_recursion<Tree>(root, TSPInfo(sz),  TSPBody(), auto_patitioner());
```

Listing 2.10: Olden tsp parallelization using `parallel_recursion`

is guaranteed to be applied to the solutions of the children subproblems generated by a given problem and that this parent problem is also an input to the method simplify extraordinarily the implementation. Concretely, the code goes from 45 SLOC using `parallel_reduce` in Listing 2.4 to 12 using `parallel_recursion`.

2.3. Evaluation

We now compare the implementation of several divide-and-conquer algorithms using `parallel_recursion`, the TBB algorithm templates and OpenMP both in terms of programmability and performance. OpenMP is not directly comparable to the skeleton libraries, as it relies on compiler support. It has been included in this study as a baseline that approaches the minimum overhead in the parallelization of applications for multicores, since the insertion of compiler directives in a program usually requires less restructuring than the definition of the classes that object-oriented skeletons use. This way the comparison of standard TBB and the

Table 2.1: Benchmarks used

Name	Description	Arity	Assoc	SLOC	Effort	V
fib	recursive computation of 43rd Fibonacci number	2	Yes	37	31707	5
merge	merge two sorted sequences of 100 million integers each	2	-	62	143004	6
qsort	quicksort of 10 million integers	2	-	71	119908	11
nqueens	N Queens solution count in 14×14 board	var	Yes	82	192727	17
treeadd	add values in binary tree with 24 levels	2	Yes	92	179387	9
bisort	sort balanced binary tree with 22 levels	2	No	227	822032	20
health	2000 simulation steps in 4-ary tree with 6 levels	4	No	346	1945582	34
tsp	traveling salesman problem on binary tree with 23 levels	2	No	370	2065129	40

`parallel_recursion` skeletons with respect to the OpenMP version helps measure the relative effort of parallelization that both kinds of skeletons imply.

The algorithms used in this evaluation are the computation of the n-th Fibonacci number from Section 2.1.1 (fib), the merge of two sorted sequences of integers into a single sorted sequence (merge), the sorting of a vector of integers by quicksort (qsort), the computation of the number of solutions to the N Queens problem (nqueens) and four tree-based Olden benchmarks [115]. The first one is treeadd, which adds values in the nodes of a binary tree. It is similar to the example in Section 2.1.2, but since the tree is binary, it is much easier to implement using TBB's `parallel_reduce`. The sorting of a balanced binary tree (bisort), a simulation of a hierarchical health system (health), and the traveling salesman problem (tsp) from Section 2.1.3 complete the list.

Table 2.1 provides the problem sizes, the number of subproblems in which each problem can be divided (arity) and whether the combination of the results of the subproblems is associative or not, or even not needed. It also shows the value of the metrics that will be used in Section 2.3.1 to evaluate the programmability for a baseline version parallelized with OpenMP. All the algorithms but nqueens and health are naturally expressed splitting each problem in two, which fits the TBB algorithm templates. Nqueens tries all the locations of queens in the i-th row of the board that do not conflict with the queens already placed in the top $i-1$ rows. Each possible location gives place to a child problem which proceeds to examine the placements in the next row. This way the number of children problems at each step varies from 0 to the board size. Health operates on a 4 -ary tree, thus four is its natural number of subproblems. The subnodes of each node are stored

in a vector. This benefits the TBB algorithm templates, as this enables using as range a `blocked_range`, which is a built-in TBB class that defines a one-dimensional iteration space, ideal to parallelize operations on vectors.

2.3.1. Programmability

The impact of the use of an approach on the ease of programming is not easy to measure. In this section three quantitative metrics are used for this purpose: the SLOC (source lines of code excluding comments and empty lines), the programming effort [63], and the cyclomatic number [91], which have been defined in Section 2.3.1.

Figure 2.1 shows the SLOC, programming effort and cyclomatic number increase over an OpenMP baseline version for each code when using a suitable TBB algorithm template (TBB) or `parallel_recursion` (pr). The statistics were collected automatically on each whole application globally. Had we tried to isolate manually the portions specifically related to the parallelization, the advantage of `parallel_recursion` over TBB would have often grown to the levels seen in the examples discussed in the preceding sections. We did not do this because sometimes it may not be clear whether some portions of code must be counted as part of the parallelization effort or not, so we measured the whole program as a neutral approach.

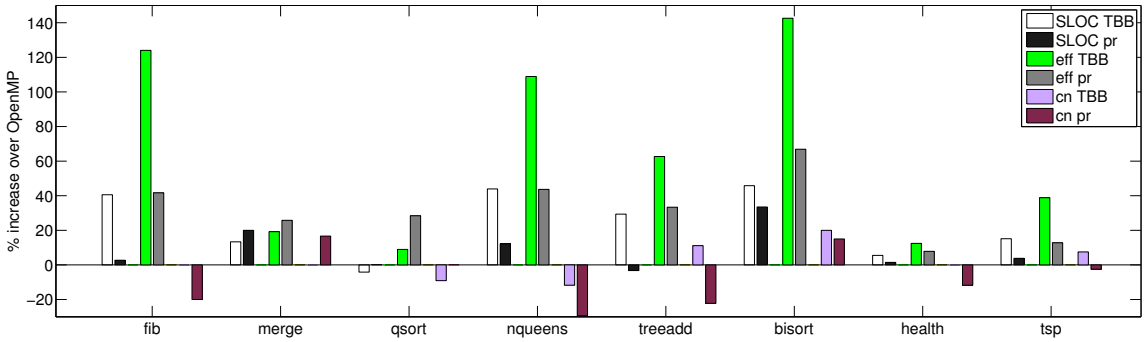

Figure 2.1: Productivity statistics with respect to the OpenMP baseline version of TBB based (TBB) and `parallel_recursion` based (pr) implementations. SLOC stands for source lines of code, eff for the programming effort and cn for the cyclomatic number.

The mostly positive values indicate that, as expected, OpenMP has the smallest programming overhead, at least when counted with SLOCs or programming effort. Nevertheless, `parallel_recursion` is the global winner for the cyclomatic number. The reason is that many of the conditionals and loops (they involve conditions to detect their termination) found in divide-and-conquer algorithms are subsumed in the `parallel_recursion` skeleton, while the other approaches leave them exposed in the programmer code more often. `parallel_recursion` requires fewer SLOC, effort and conditionals than the TBB algorithm templates in all the codes but merge and qsort. According to the programming effort indicator, programs parallelized with the TBB templates require 64.6% more effort than OpenMP, while those based on `parallel_recursion` require on average 33.3% more effort than OpenMP. This is a reduction of nearly 50% in relative terms. Interestingly, the situation is the opposite for merge and qsort, in which the average effort overhead over the OpenMP version is 13.4% for the codes that use `parallel_for` and 30.1% for the `parallel_recursion` codes. These are the only benchmarks in which there is no need to combine the result of the solution of the problems: they only require the division in subproblems that can be solved in parallel. They are also the two benchmarks purely based on arrays, where the concept of Range around which the TBB algorithm templates are designed fits better. Thus when these conditions hold, we may prefer to try the standard TBB skeletons.

2.3.2. Performance

The performance of these approaches is compared now using the Intel icpc compiler V 11.0 with optimization level O3 in two platforms. One is a server with 4 Intel Xeon hexa-core 2.40GHz E7450 CPUs, whose results are labeled with X. The other is an HP Integrity rx7640 server with 8 dual-core 1.6 GHz Itanium Montvale processors, whose results are labeled with I. Figures 2.2 to 2.9 show the performance of the three implementations of the benchmarks on both systems. Automatic partitioning is used in the standard TBB and `parallel_recursion` based codes. Fib and nqueens use little memory and thus scale well in both systems. The scaling of the other benchmarks is affected by the lack of memory bandwidth as the number of cores increases, particularly in our Xeon-based system, whose memory bandwidth is up to 5 times smaller than that of the rx7640 server when 16 cores are used. This

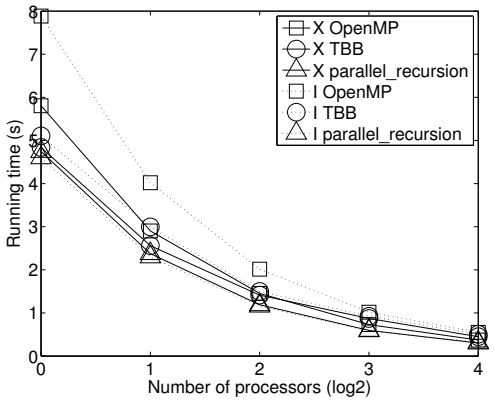

Figure 2.2: Performance of fib

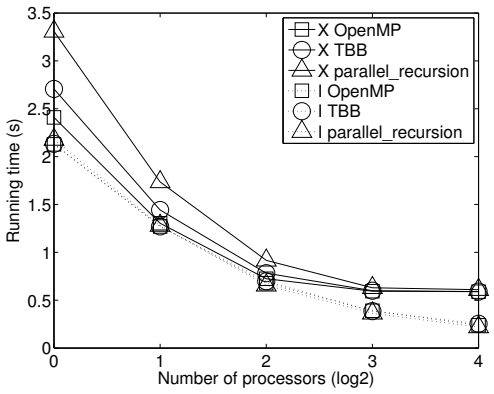

Figure 2.3: Performance of merge

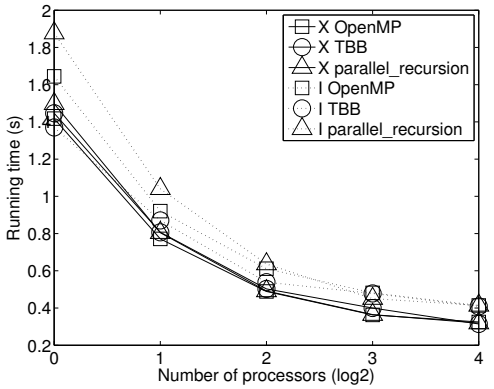

Figure 2.4: Performance of quicksort

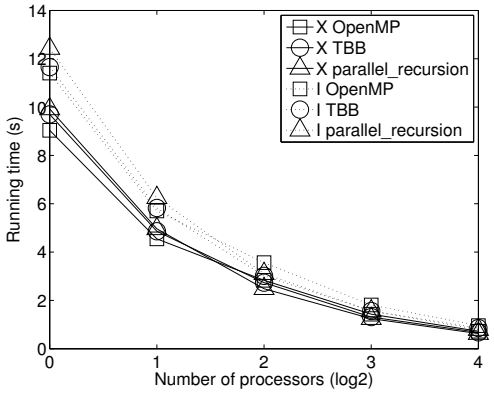

Figure 2.5: Performance of nqueens

results in small to null performance improvements when we go from 8 to 16 cores in this system. Benchmark health is also affected by very frequent memory allocations with `malloc` that become a source of contention due to the associated lock.

Since `parallel_recursion` is built on top of the TBB one could expect its codes to be slower than those based on `parallel_for` or `parallel_reduce`. This is not the case because our template is built directly on the low level task API provided by the TBB. Also, it follows different policies to decide to spawn tasks and has different synchronization and data structure support requirements as we have seen. This makes it possible for `parallel_recursion` to be competitive with the native TBB version, and even win systematically in benchmarks like fib. In other benchmarks like merge in the Xeon or quicksort in the Itanium `parallel_recursion` is non

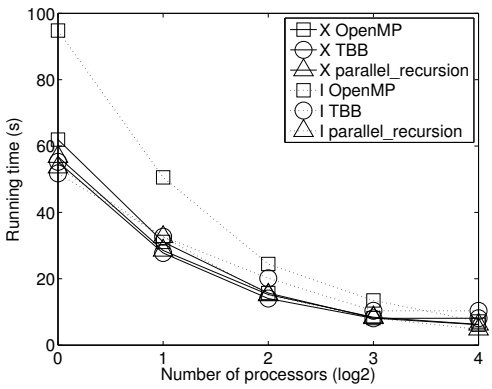

Figure 2.6: Performance of treeadd

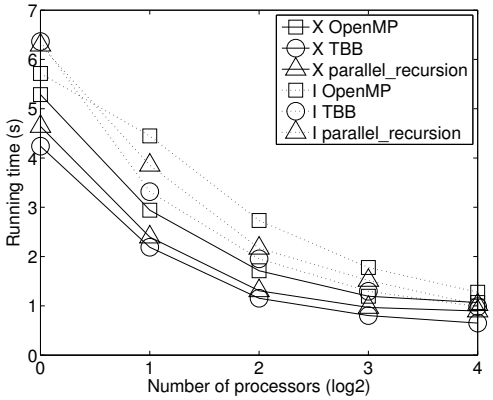

Figure 2.7: Performance of bisort

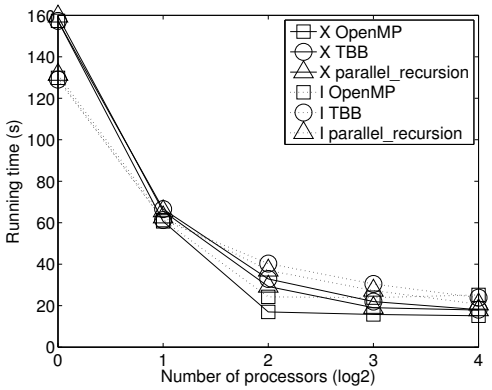

Figure 2.8: Performance of health

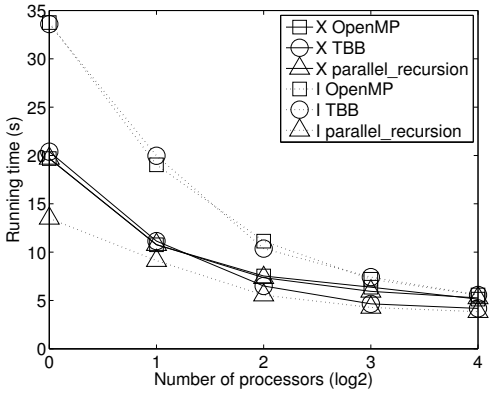

Figure 2.9: Performance of tsp

negligibly slower than the standard TBB when few cores are used, but the difference vanishes as the number of cores increases. The slowdowns of `parallel_recursion` in these two codes are due to operations repeated in each invocation to `child` to generate a subproblem. Generating at once a vector with all the children tasks could avoid this. Evaluating this option is part of our future work. The behavior of tsp in the Itanium is due to the compiler, as with g++ 4.1.2 with the same flags the performance of all the implementations is very similar. Over all the benchmarks and numbers of cores, on average `parallel_recursion` is 0.3% and 19.7% faster than the TBB algorithm templates in the Xeon and in the Itanium, respectively. If tsp is ignored in the Itanium due to its strange behavior, `parallel_recursion` advantage is still 9% in this platform. Its speedups over OpenMP are 2.5% and 30.5% in the Xeon and the Itanium, respectively; 21.4% in the latter without tsp.

We also experimented with the partitioners that allow to control manually the subdivision of tasks in the runs with 16 cores. With the best parameters we found, standard TBB based codes were on average 6.5% faster than the `parallel_recursion` based ones in the Xeon, while `parallel_recursion` continued to lead the performance in the Itanium platform by 8%, or 2.4% if tsp is not counted.

2.4. Related work

While TBB is probably the most widespread library of skeletal operations nowadays, it is not the only one. The eSkel library [34] offers parallel skeletal operations for C on top of MPI. Its API is somewhat low-level, with many MPI-specific implementation details. Since C is not object oriented, it cannot exploit the advantages of objects for encapsulation, polymorphism, and generic programming where available, as is the case of C++. A step in this direction was Muesli [30], which is also oriented to distributed memory, being centered around distributed containers and skeleton classes that define process topologies. Muesli relies on runtime polymorphic calls, which generate potentially large overheads. This way [75] reports 20% to 100% overheads for simple applications. Lithium [6] is a Java library oriented to distributed memory that exploits a macro data flow implementation schema instead of the more usual implementation templates, but it also relies extensively on runtime polymorphism. Quaff [46] avoids this following the same approach as the TBB and our proposal, namely relying on C++ template metaprogramming to resolve polymorphism at compile time. Quaff's most distinctive feature is that it leads the programmer to encode the task graph of the application by means of type definitions which are processed at compile time to produce optimized message-passing code. As a result, while it allows skeleton nesting, this nesting must be statically defined, just as type definitions must be. Thus tasks cannot be generated dynamically at arbitrary levels of recursion and problem subdivision as the TBBs do. This is quite sensible, since Quaff works on top of MPI, being mostly oriented to distributed memory systems. For this reason its `scm` (split-compute-merge) skeleton, which is the most appropriate one to express divide-and-conquer algorithms, differs substantially from the TBB standard algorithm templates and `parallel_recursion`.

2.5. Conclusions

We have reviewed the limitations of the skeletal operations of the TBB library to express the divide-and-conquer pattern of parallelism. This analysis has led us to design a new algorithm template that overcomes these problems. We have also implemented it on top of the task API of the TBB so that it is compatible with all the TBB library and it benefits from the load balancing of the TBB scheduler. Our implementation uses template metaprogramming very much as the standard TBB in order to provide efficient polymorphism resolved at compile time.

The examples we examined and an evaluation using several productivity measures indicate that our algorithm template can often improve substantially the programmer productivity when expressing divide-and-conquer parallel problems. As for performance, our proposal is on average somewhat faster than the TBB templates when automatic partitioning is used. There is not a clear winner when the granularity of the parallel tasks is adjusted manually.

Chapter 3

Parallel skeleton for domain-based irregular algorithms

In this chapter we turn our attention to algorithms that genuinely fit the amorphous data-parallel paradigm [76]. Our proposal to simplify their parallelization, whose complexity has been discussed in Section 1.2, consists in a parallel skeleton based on the abstraction of a domain defined in terms of some property of the problem elements. This domain is used both to partition the computation, by assigning the elements of different subdomains to different parallel tasks, and to avoid conflicts between these tasks, by checking whether the accessed elements are owned by the subdomain assigned to the task. Our proposal applies a recursive scheduling strategy that avoids locking the partitions generated, instead delaying work that might span partitions until later in the computation. Among other benefits, this approach promotes the locality in the parallel tasks, avoids the usage of locks, and thus the contention and busy waiting situations often related to them, and provides guarantees on the maximal number of abortions due to conflicts between parallel tasks during the execution of an irregular algorithm. Following the description of this strategy and its requirements, an implementation as a C++ library is also described and evaluated. The flexibility of our proposal will also allow to show how easy it is to modify relevant implementation decisions such as the the domain partitioning algorithm or the data structures used and measure their impact on performance.

3.1. Domain-Based Parallel Irregular Algorithms

Many algorithms that operate on irregular data structures have a workflow based on the processing of a series of elements belonging to the irregular structure, called *workitems*. The elements that need some processing done on them are stored in a generic *worklist*, which is updated as the algorithm runs when new workitems are found. The pseudocode in Listing 3.1 shows the general workflow of these algorithms. Line 1 fills the initial worklist with elements of the irregular structure. Any irregular structure could fit our generic description of the pseudocode and our subsequent discussion. In what follows we will use the term graph, as it is a very generic irregular data structure and many others can be represented as graphs too. Some algorithms start with just one root element, while others have an initial subset of the elements or even the full graph. The loop in Lines 2 to 6 processes each element of this worklist. Line 3 represents the main body of the algorithm being implemented. If this processing results in new work being needed, as checked in Line 4, it is added to the worklist in Line 5. This is repeated until the worklist is empty.

Some characteristics can be identified in these kinds of algorithms. An important one is whether the workitems must be processed in some specific order. A study in [66] on the characteristics and behavior of the ordered and non-ordered versions of a series of algorithms shows that the unordered versions present more parallelism and behave better in terms of scalability due to the complexity inherent to the efficient implementation of the ordering. Following its recommendation to use unordered algorithms when possible, these algorithms are the scope of our work.

The workflow of unordered algorithms can be parallelized by means of having different threads operating on different elements of the worklist, provided that no

```
1  Worklist wl = get_initial_elements_from(graph);
2  foreach(element e in wl) {
3    new_work = do_something(e);
4    if(new_work != null)
5      wl.push(new_work);
6  }
```

Listing 3.1: Common pseudocode for an algorithm that uses irregular data structures

conflicts appear during the simultaneous processing of any two workitems. If the algorithm requires the workitems to be processed in some specific order, for example the one in which they are stored in the worklist, special measures should be taken, which are not covered in this work.

The workitems found in irregular algorithms usually have properties (in the following, property refers to a data item, such as for example a data member in a class) defined on domains, such as names, coordinates or colors. Therefore a sensible way to partition the work in an irregular algorithm is to choose a property of this kind defined on the workitems, and classify them according to it. Specifically, the domain of the property would be divided in subdomains and a parallel task would be launched to process the workitems of each subdomain. In principle any property would do, but in practice a few characteristics are required in order to attain good performance. If no intrinsic property of the problem meets them, an additional property satisfying them should be defined in the workitems for the sake of a good parallelization following this scheme.

The first characteristic is that the domain of the property should be divisible in at least as many subdomains as hardware threads are available in the system, the subdomains being as balanced as possible in terms of workitems associated. In fact, it would be desirable to generate more subdomains than threads in order to provide load balancing by assigning new subdomain tasks to threads as they finish their previous task. Second, if the processing of a workitem generates new workitems, it is desirable that the generated workitems belong to the same subdomain as their parent. We call this characteristic, which depends also on the nature of the operation to apply on the workitems, affinity of children to parents. If this were not the case, either the rule of ownership of the workitems by tasks depending of the subdomain they belong to would be broken, or intertask communication would be required to reassign these workitems to the task that owns their subdomain. Third and last, there is the proximity characteristic, that is, that the larger the similarity in the values of the chosen property, the shorter the distance between the associated workitems in the graph.

Very often the processing of a workitem requires accessing part of its neighborhood in the graph. If some element(s) in this neighborhood belong to other tasks the processing is endangered by potential parallel modifications by other threads.

Nevertheless, if all the elements required belong to the subdomain of the workitem that started the processing, everything is owned by the task for that subdomain and the processing can proceed successfully. This way, if the rule of ownership is fulfilled, i.e, all the elements of the graph that belong to a certain subdomain are owned by the same task, subdomains can be used not only to partition work, but also to identify potential conflicts. The process will be besides efficient if the property chosen to define the work domains implies proximity for the elements that belong to the same subdomain. For this reason, in algorithms where the processing of a workitem requires accessing its neighborhood, the characteristics of the affinity of children to parents and proximity are very desirable.

3.2. A parallelization scheme based on domains

The data partitioning coupled with data-centric work assignment we have just presented to improve the parallel execution of irregular algorithms is a basic idea that can be put into practice in very different ways. We propose here a scheme based on the recursive subdivision of a *domain* defined on the elements of the irregular data structure, so that the workitems of each subdomain are processed in parallel, and the potential conflicts among them are exclusively detected and handled using the concept of membership of the subdomain. Locality of reference in the parallel tasks is naturally provided by the fact that most updates in irregular applications are usually restricted to small regions of the shared heap [76][85]. Our scheme further reinforces locality if the domain used in the partitioning fulfills the proximity characteristic, so that the elements associated to a subdomain, and therefore a task, are nearby. The processing of the workitems begins in the lowest level of subdivision, where there is the maximum number of subdomains, and therefore parallel tasks. The workitems that cannot be processed within a given subdomain, typically because they require manipulations of items associated to other subdomains, are later reconsidered for processing at higher levels of decomposition using larger subdomains. We now explain in detail our parallelization method, illustrated in Figure 3.1. This figure shows a mesh of triangles, which can be stored in a graph where each node is a triangle and the edges connect triangles which are next to each other in the mesh. The big dots represent the possible limits of the subdomains. In

this case, the domain chosen is defined on the coordinates of the triangles.

3.2.1. Recursive subdivision

An algorithm starts with an initial worklist, containing nodes from the whole graph domain, as shown in the initial step in Figure 3.1. Before doing any processing, the domain can be recursively subdivided until it reaches an adequate level, that is, until there are enough subdomains to get a good balance between potential parallelism and hardware resources in the system, namely processing cores. The domain decomposition algorithm chosen can have a large impact on the performance achieved. The reason is that the size of the different parallel tasks generated, which is critical for the load balancing, and the shape of the subdomains they operate on, which influences the number of potential conflicts during the parallel processing, largely depend on it. Over-decomposition, i.e., generating more subdomains than cores, can be applied with the aim of implementing load balancing between the cores by means of work-stealing by idle cores. The subdivision of the domain implicitly partitions both the graph and the worklist. This logical partitioning can optionally give place to a physical partitioning. That is, the graph and/or the worklist can be partitioned in (mostly) separate data structures so that each one corresponds to the items belonging to a given subdomain and can be manipulated by the associated task with less contention and improved locality. We talk about mostly separate structures because for structures such as the graph, tasks should be able to access portions assigned to other tasks. It is up to the implementation strategy to decide which kind of partitioning to apply to each data structure. In our abstract representation, for simplicity, we show 2 subdivisions to get 4 different subdomains, in Steps 1 and 2. Then, in Step 3, a parallel task per subdomain is launched, whose *local worklist* contains the elements of the global worklist that fall in its subdomain. During the processing of each workitem two special events can happen: an access to an element outside the local subdomain, and the generation of new workitems to process. We describe the approach proposed for these two situations in turn.

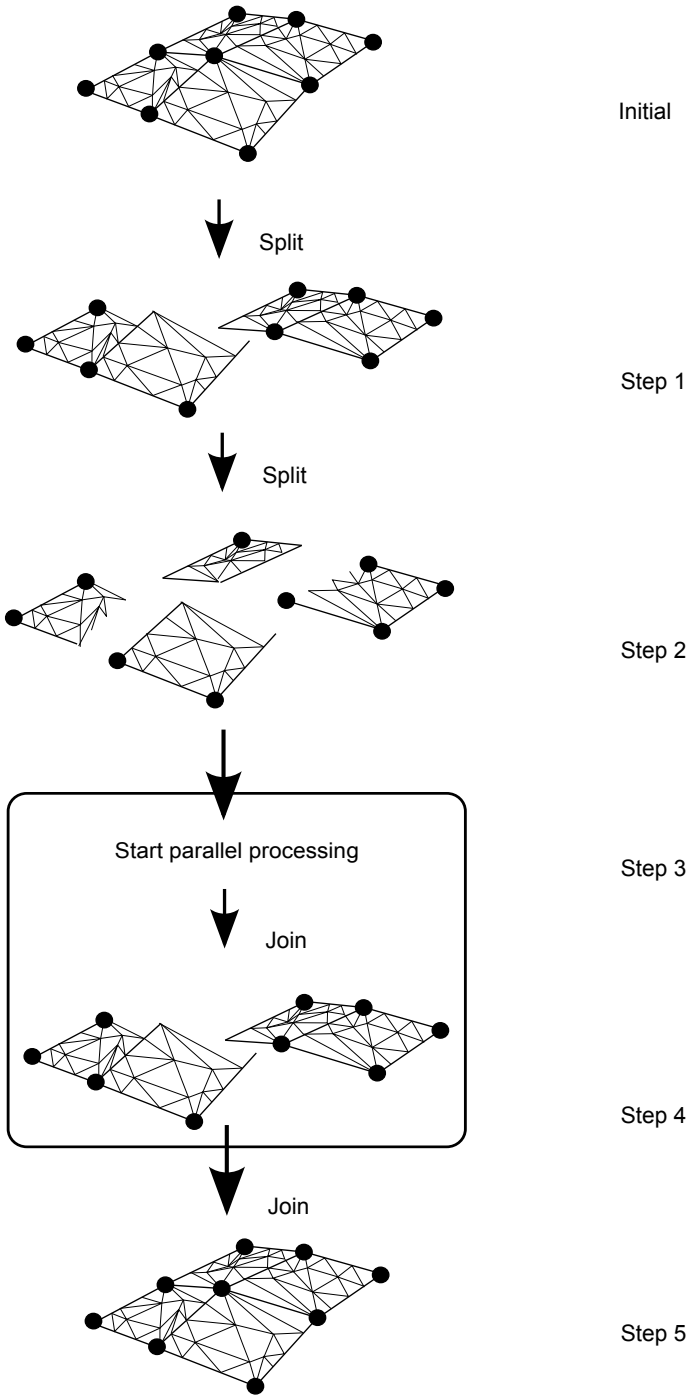

Figure 3.1: Structure of the domain-based parallelization of irregular algorithms

3.2.2. Conflict detection

In some algorithms the processing of a workitem only requires access to the data specifically associated to the workitem. Nevertheless, in many other algorithms, for example in all those that modify the graph, the processing associated to a workitem requires accessing —either reading or even modifying— a given set of edges and nodes around it, which is called the neighborhood. The neighborhood is often found dynamically during the processing and its extent and shape can vary for different workitems. This way we have to deal with the possibility that the neighborhood required for the processing of a workitem reaches outside the subdomain of the associated task. Accessing an element outside the local the subdomain owned by the current task is a risk, since it could be in an inconsistent state or about to be modified by another task. Therefore we propose that whenever a new element in the neighborhood of a workitem is accessed for the first time, its ownership by the local domain is checked. If the element belongs to the domain, the processing proceeds. Otherwise there is a potential conflict and the way to proceed depends on the state of our processing. Following the terminology in [92], an operation is said to be cautious if it reads all the elements of its neighborhood before it modifies any of them. Therefore, if the processing fulfills this property, all it needs to do when it finds an element owned by another task is to leave, as no state of the problem will have been modified before. If the operation is not cautious, rollback mechanisms would have to be implemented to undo the modifications performed.

When a task fails to complete the processing of a workitem because part of its neighborhood falls outside its domain, it puts the workitem in a pending list to be processed later, which is different from the local worklist of workitems to be processed. The processing of this pending list will be discussed in Section 3.2.4.

Notice that the more neighbors a node has, the higher the chances all its neighborhood does not fit in a single subdomain. For this reason nodes with a large number of neighbors will tend to generate more conflicts, and thus lower performance, depending on the domain and decomposition chosen. The programmer could avoid this problem by choosing a domain with a subdivision algorithm that fits this kind of graphs for the specific problem she is dealing with. For example the domain and splitting algorithm could be designed such that nodes with many neighbors always,

or at least often, fit in the same subdomain with their neighbors.

3.2.3. Generation of new workitems

When the processing of a workitem generates new workitems, they are simply added to the local worklist, assuming the property used for the definition of the domains fulfills the characteristic of affinity of children to parent discussed above. If the property does not guarantee this affinity, the newly generated workitems can be still added to the local worklist, but then the ownership by the current subdomain of each workitem must be checked as it is retrieved from the local worklist. This can be made automatically by the algorithm template, or it can be left as a responsibility for the operation provided by the user to process the workitems. In either case, generated workitems not belonging to the local domain will be detected and their processing delayed to later stages, exactly like workitems whose neighborhood extends outside the subdomain owned by the task.

There is another option for the processing of non local workitems that can be very interesting: pushing them in the worklists associated to their domains, so their processing starts as soon as possible. This alternative is particularly suitable and easy to apply if it is the algorithm template who is left in charge of checking the ownership of the newly generated workitems and acting accordingly. This behavior would allow to parallelize following our proposal algorithms that start with a few workitems —even a single one— and expand their working set from there. A representative example of this kind of algorithms are graph searches.

3.2.4. Domain merging

When there are no more elements left in the local worklist of a subdomain task, the task finishes and the processing can proceed to the immediately higher level of domain subdivision, as shown in Step 4 in Figure 3.1. The internal implementation of the change of level of processing can be synchronous or not. In the first case, the implementation will wait for all the tasks for the subdomains of a given level to finish before building and launching the tasks for all the domains in the immediately upper level. In an asynchronous implementation, whenever the two children subdomains

of a parent domain finish their processing, a task is built and sent for execution for the parent domain. In either case, each two children domains of a given parent subdomain are rejoined, forming that parent domain, and the pending lists generated in the children subdomains are also joined forming the worklist of the task for the parent domain. An efficient implementation of this scheme should perform the aforementioned merging, and also schedule for execution the task associated to the parent domain, in one of the cores involved in the execution of the children tasks, the purpose being maximizing the exploitation of locality. When it runs, the task associated to the parent domain tries to process the workitems whose processing failed in the children domains for the reasons explained above. The task will succeed in the processing of those workitems whose neighborhood did not fit in any of the children subdomains, but which fits in the parent domain. Typically the processing of some workitems will fail again because the neighborhood for their processing falls also outside this domain. These workitems will populate the pending list of the task. This process takes place one level at a time as the processing returns from the recursive subdivision of the initial domain. This way, the tasks for all the joined regions —except the topmost one— are processed in parallel. This is repeated until the original whole domain is reached, and the remaining elements are processed, which is depicted as the final Step 5 in Figure 3.1.

3.2.5. Discussion

As we have seen, the proposed scheme avoids the need of locks not just on the elements of the graph, but even on the subdomains and implied partitions generated, therefore avoiding the busy waiting and contention problems usually associated to this synchronization mechanism. Also, its strategy to deal with conflicts provides an upper bound for the number of attempts to process workitems whose neighborhood extends outside the partition assigned to their tasks. Those workitems are considered at most once per level of subdivision of the original domain, rather than being repetitively reexamined until their processing succeeds. Both characteristics are very desirable, particularly as the number of cores, and therefore parallel tasks and potential conflicts, increases. This strategy has though the drawback of eventually serializing the processing of the last elements. But because of the rejoining process aforementioned, which tries to parallelize as much as possible the processing of

the workitems whose processing failed in the bottom level of subdivision of the domain, the vast majority of the work is performed in parallel. In fact, as we will see in Section 3.5, in our tests we observed that only a very small percentage of the workitems present conflicts that prevent their parallel processing. This also confirms that optimistic parallelization approaches like ours are very suitable for irregular applications [79][78].

3.3. The library

We have developed a library that supports our domain-based strategy to parallelize irregular applications in shared- memory systems. It provides a series of templates that cover the basic requirements to implement a wide range of irregular algorithms. First there is a graph template class, which can be used to store the data for the algorithm. Next, a container for supporting the worklists is included, which is defined closely integrated with the graph class. Common 1-dimensional and 2-dimensional domain classes are provided for convenience. Finally, the central template function of the library is `parallel_domain_proc`. It gets a graph, a worklist, a domain, and an operation provided by the user to process a workitem —as a functor, function pointer or lambda function. The programmers can use the types provided by the library, derive from them or implement their own from scratch, as long as they meet certain interface requirements. The following subsections introduce the elements of the library and these requirements.

3.3.1. Graph

The graph is the structure that typically contains the workitems and the relations among them. The `Graph` class provided by our library acts as a container for `Node` and `Edge` objects, which in turn contain the actual data as defined by the programmers. The library does not enforce the graph to be connected, it can be made of disjoint clusters. Also, directed and undirected versions of the graph are provided.

In our current implementation of the library, the `Graph` is only manipulated by

the function provided by the user to perform the operation on each workitem, so the interface can be up to the user. As a result, the only requirement for an alternative user provided graph class is to support the simultaneous addition and deletion of nodes and edges if the algorithm to implement modifies the structure of the graph. The `Graph` class provided fulfills this condition. The reason for this requirement is that, as we will see, our algorithm template does not partition physically the graph; therefore all the parallel tasks access the same graph object. Locks to access each element are not required, though.

3.3.2. Worklist

The straightforward approach would be to use worklists composed of the workitems to process or pointers to them. Unfortunately, many irregular algorithms destroy workitems during the processing of other workitems. Searching the worklist for a potential workitem to remove each time one is destroyed is costly unless the worklist has some ordering, but then inserting new workitems would be more expensive. If the destruction were implemented as a logical mark in the workitem indicating that it no longer must be processed, worklists based on pointers to workitems would still work. Still, for some algorithms the memory overhead of not deallocating no longer needed data items is overwhelming for medium to large data problem sizes. Therefore, an alternative approach is required.

Our graph classes keep two internal containers of pointers to edges and nodes, respectively, to manage the graph components. When a graph component is deallocated, the pointer in the respective container is zeroed. The `Edge` and `Node` base classes of our graphs contain a pointer back to the pointer pointing them in the corresponding container, so that no search is needed to manipulate it when needed. Pointers to locations in the containers of elements of these graph classes are thus ideal components for the worklists, as they can be safely dereferenced to learn whether the element still exists. This way, the `Worklist` type provided for convenience by our library, and expected by our algorithm template, is composed of these pointers. This allows our algorithm template to detect and automatically skip deallocated workitems and proceed to process the other ones. Let us notice that this design of the worklist is also useful for the sequential implementations of the irregular

algorithms, which face the same problem.

Additionally, our graph classes provide a helper method `add_to _worklist` that takes as input a worklist and a graph element pointer (either node or edge). This method adds to the worklist the pointer to the location in the graph container associated to the element. This way the process is transparent to the user, who simply knows that workitems must be added using this method, not needing thus any information on the internals of the classes.

Again, programmers can use any worklist type as long as it fulfills two conditions. The first one is that a single derefence of its items must provide the pointer to the actual workitem to process, or a nil pointer if the workitem no longer exists. The second requirement is to implement the basic interface shown in Listing 3.2 to insert, read and remove elements in the back of the list, respectively.

3.3.3. Domain

The domain is used to classify each element of the graph and the worklist in a different group, depending on its data. This allows the `parallel_domain_proc` algorithm template to create several units of work to be run in parallel and avoid conflicts among tasks.

A class that defines this domain must be provided to our skeleton, with methods to check whether an element falls inside a domain, to check whether a domain is

```
1 void push_back(const value_type& v)
2 value_type back()
3 void pop_back()
```

Listing 3.2: Required interface for the `Worklist` class

```
1 bool contains(Element* e)
2 bool is_divisible()
3 void split(Domain& s1, Domain& s2)
```

Listing 3.3: Required interface for the `Domain` class

splittable, and to perform this splitting. The interface for this class is shown in Listing 3.3. We can see that, though the strategy described in this chapter involves merging children subdomains, no method is required for this. The reason is that our algorithm template stores the input and intermediate domain objects generated during the recursive subdivision in order to use them when the processing returns from the bottom level. Our library includes domain template classes for linear and bidimensional domains. The template arguments are in both cases the types of the coordinate to use and the workitem to check. The user will redefine `contains` in a derived class in order to specify how to extract the property to check from the workitem if it does not have the default name assumed by the class template.

3.3.4. Operation

The last element required by our algorithm template is the operation to be done on each element of the worklist, which is not part of the library, as it must be always provided by the user. It takes the form

`void op(Element * e, Worklist& wl, Domain& s).`

These parameters are the current workitem/element of the graph to process, the local worklist and the current subdomain, which will be provided by our algorithm template in each invokation. When the domain is subdivided, the initial worklist is divided in different worklists too, thus each thread has it own local worklist. The operation can be implemented as a functor object, a function pointer or a C++11 lambda function. When accessing the neighbors of a node, the operation is responsible for checking whether these neighbors fall outside the current domain, usually using the methods of the `Domain` object. The processing of an element may create new elements that should be processed. If so, the operation must add them to the subdomain task local worklist `wl`. When processing an element requires access to a neighbor that falls in other subdomain, the operation must throw an exception of a class provided by our library. This exception, which is captured by our algorithm template, tells the library to store the current workitem in the pending list, so it can be processed when the subdomains are joined.

The programmability of operations can be improved by extending the domain

class used with a method that checks whether the input element belongs to the domain, and throws this exception if this is not case. The domain template classes provided by our library offer a method called `check` with this functionality.

3.3.5. parallel_domain_proc skeleton

The kernel of our library is the algorithm template that implements the parallelization approach described in this chapter, which is

```
1 void parallel_domain_proc<bool redirect=false>
2               (Graph, Worklist, Domain, Operation)
```

where the parameters are objects of the classes explained in the previous sections, or any classes that fulfill the corresponding requirements explained. This function is in charge of the domain splitting process, task creation and management, splitting and merging the worklists, getting elements from them to run the operation, and adding new elements to the pending worklists when out-of-domain exceptions are thrown. It deserves to be mentioned that this skeleton partitions physically the worklists, so that each parallel task has its own separate worklist object. Thanks to the physical partitioning, the worklists need not support simultaneous accesses from parallel tasks. However, the fact that these containers are extensively read and modified during the parallel execution makes their design important for performance. Nonetheless, the partition of the graph is only logical, that is, it is virtually provided by the existence of multiple subdomains, there being a single unified graph object accessed by all the tasks. This implies, as we explained in Section 3.3.1, that our library graphs can be safely read and updated in parallel, as long as no two accesses affect the same element simultaneously —unless they are all reads.

First, the domain is subdivided recursively, creating several leaf domains. The subdivision process for a domain stops when either it is not divisible —domains must provide a method to inform whether they are divisible, as we will soon see— or `parallel_domain_proc` decides there are enough parallel tasks for the hardware resources avaiable. This is the same approach followed by popular libraries like Intel Threading Building Blocks [112]. Our initial implementation, evaluated in Section 3.5, always partitions the input domain until there are at least two subdo-

mains per hardware thread available. The existence of more tasks than threads is used internally by our implementation to balance the load among the threads, as they take charge of new tasks as they finish the previous one. The initial workload is distributed among these subdomains, assigning each workitem to a subdomain depending on the value of its data. Then a task is scheduled for each subdomain, which will process the worklist elements belonging to that subdomain and which will have the control on the portion of the graph that belongs to that domain.

The boolean template parameter `redirect` controls the behavior of the algorithm template with respect to the workitems extracted from the local worklist whose processing fails (due to out-of-domain exceptions). When `redirect` is `false` — which is its default—, they are simply pushed in the tasks's pending list. When it is `true`, the behavior depends on the state of the task associated to the workitem subdomain at the bottom level of subdivision. If this task or a parent of it is already running, the workitem is stored in the pending list of the task that generated it, so there is not redirection. Otherwise, it is stored in the local worklist of the task that owns its subdomain, and if this task were not scheduled for execution, it is enqueued in the list of tasks waiting to be run. To facilitate the redirection of workitems, this configuration of the algorithm template does not schedule for execution tasks whose worklists are empty. Also, non-bottom tasks are run when either their two children finished — as usual— or when there are no lower level tasks running or waiting to run, as the tasks for some subdomains could never run. Notice that this boolean flag is a performance hint, as all the workitems will be correctly processed no matter which is its value. Redirection will mostly benefit algorithms in which the items in the initial worklist belong to a limited number of bottom level subdomains, and where the processing will gradually evolve to affect more graph subdomains.

The skeleton populates the worklist of the tasks associated to non-bottom subdomains with the workitems found in the pending lists of their respective children. This way, when the skeleton launches them for execution, they try to process the elements that could not be processed in their children. This process happens repetitively until the root of the tree of domains —i.e., the initial domain provided by the user—, is reached.

3.4. Tested algorithms

We tested our library with several benchmarks, both in terms of programmability and performance. As of now our library does not provide mechanisms to rollback computations. Therefore all the algorithms tested are cautious —i.e., they do not need to restore data when an operation fails due to the discovery of a potential conflict. Now a brief explanation of each benchmark follows.

3.4.1. Boruvka

Boruvka's algorithm computes the minimal spanning tree through successive applications of edge-contraction on the input graph. In edge-contraction, an edge is chosen from the graph and a new node is formed with the union of the connectivity of the incident nodes of the chosen edge, as shown in Figure 3.2. In the case that there are duplicate edges, only the one with least weight is carried through in the union. Figure 3.2 demonstrates this process. Boruvka's algorithm proceeds in an unordered fashion. Each node performs edge contraction with its lightest neighbor. This is in contrast with Kruskal's algorithm where, conceptually, edge-contractions are performed in increasing weight order. The pseudocode for the algorithm is shown in Listing 3.4.

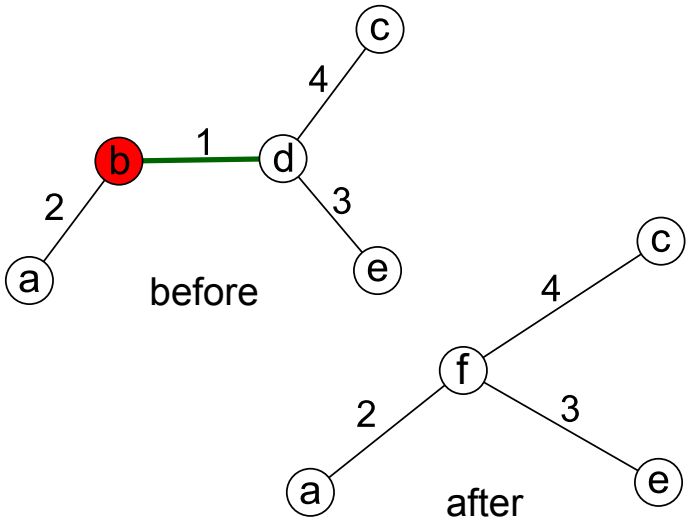

Figure 3.2: Example of an edge contraction of the Boruvka algorithm

```
1 Graph g = read_graph();
2 Forest mst = g.nodes();
3 Worklist wl = g.nodes();
4 foreach(Node n in wl) {
5   Node m = min_weight(n, g.get_out_edges(n));
6   Node l = edge_contract(n, m);
7   mst.add_edge(n, m);
8   wl.add(l);
9 }
```

Listing 3.4: Pseudocode of the Boruvka minimum spanning tree algorithm

First, it reads the graph in Line 1, and fills the worklist with all the nodes of the graph. The nodes of the initial MST are the same as those of the graph, and they are connected in the loop in Lines 4 to 9. For each node, the minimum weighted node from it to its neighbors is selected in Line 5. Then, in Line 6, this edge is contracted: it is removed from the graph, added to the MST in Line 7, and one node represents now the current node and its neighbor. This new node is added to the worklist in Line 8.

The parallelism available in this algorithm decreases over time. At first, all the nodes whose neighborhoods does not overlap can be processed in parallel, but as it proceeds the graph gets smaller, so there are less nodes to be processed.

3.4.2. Delaunay mesh refinement

This benchmark is an implementation of the algorithm described in [29]. A 2D Delaunay mesh is a triangulation of a set of points that fulfills the condition that for any triangle, its circumcicle does not contain any other point from the mesh. A mesh refinement has the additional constraint of not having any angle less than 30 degrees. This algorithm takes as input an Delanuay mesh, which may contain triangles not meeting the constraint, which are called bad triangles. As output, it produces a refined mesh by iteratively re- triangulating the affected positions of the mesh. Figure 3.3 shows an example of a refined mesh.

The pseudocode for the algorithm is shown in Listing 3.5, and works as follows.

```
1 Mesh m = read_mesh();
2 Worklist wl = m.bad_triangles();
3 foreach(Triangle t in wl) {
4   Cavity c = new Cavity(t);
5   c.expand();
6   c.retriangulate();
7   m.update_mesh(c);
8   wl.add(c.bad_triangles());
9 }
```

Listing 3.5: Pseudocode of the Delaunay mesh refinement algorithm

Line 1 reads a mesh definition and stores it as a `Mesh` object. From this object, we can get the bad triangles as shown in Line 2, and save them as an initial worklist in `wl`. The loop between Lines 3 and 9 is the core of the algorithm. Line 4 builds a `Cavity`, which represents the set of triangles around the bad one that are going to be retriangulated. In Line 5 this cavity is expanded so it covers all the affected neighbors. Then the cavity is retriangulated in Line 6, and the old cavity is substituted with the new triangulation in Line 7. This new triangulation can in turn have created new bad triangules, which are collected in Line 8 and added to the worklist for further processing.

The triangles whose neighborhood does not overlap can be processed in parallel, because there will be no conflicts when modifying them. When the algorithm starts, chances are that many of this bad triangles can be processed in parallel.

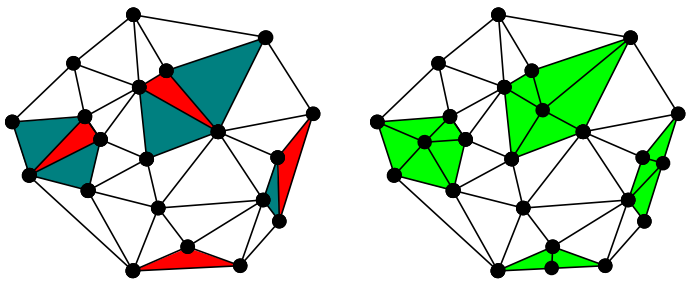

Figure 3.3: Retriangulation of cavities around bad triangles

3.4.3. Graph labeling

Graph component labeling involves identifying which nodes in a graph belong to the same connected cluster. The algorithm and serveral GPU implementations are explained in [67]. We have used the CPU version, whose pseudocode is shown in Listing 3.6. The algorithm initializes the colors of all vertices to distinct values in Lines 6 to 9. It then iterates over the vertex set V and starts the labeling procedure

```
1  map<vertex, int> color; // Color for each vertex
2  map<vertex, bool> process; // Stores whether each vertex requires more processing
3  Graph g = readGraph();
4  Worklist wl = g.nodes();
5
6  foreach(Node n in g.nodes) {
7    color[n] = i;
8    process[n] = true;
9  }
10
11 foreach(Node n in wl) {
12   if(process[n]) {
13     do_process(n);
14   }
15 }
16
17 do_process(Node n) {
18   process[n] = false;
19   foreach(edge e in n.edges()) {
20     if(color[e.source] < color[e.destination]) {
21       color[e.destination] = color[e.source];
22       do_process(e.destination);
23     }
24     else if(color[e.source] > color[e.destination]) {
25       color[e.source] = color[e.destination];
26       restart loop from start of the list;
27     }
28   }
29 }
```

Listing 3.6: Pseudocode of the graph labeling algorithm

for all vertices that have not been labelled yet, in Lines 11 to 15. The labeling procedure iterates over the edge set of each vertex, comparing in Line 20 its color value with that of its neighbors. If it finds that the color value of a neighbor is greater, it sets it to the color of the current vertex and recursively calls the labeling procedure on that neighbor in Lines 21 and 22. If the neigbor has a lower color value, Lines 25 sets the color of the current vertex to that of the neighbor and Line 26 starts iterating over the list of edges of the node from the beginning again.

3.4.4. Spanning tree

This algorithm computes the spanning tree of an unweighted graph. It starts with a random root node, and it checks its neighbors and adds to the tree those not already added. The processing continues from each one of these nodes, until the full set of nodes has been checked and added to the graph. This algorithm is somewhat different from the ones previously explained, because it starts with just one node in the worklist, while the others have an initial worklist with a set of nodes distributed over all the domain of the graph. The pseudocode is shown in Listing 3.7.

The steps aformentioned are located as follows: Line 1 reads the graph, and Lines 2 and 3 create an empty tree and a worklist with a random node respectively. The loop in Lines 5 to 10 adds to the MST the neighbors of the current node that are not already in it, and then inserts such neighbor in the worklist for further

```
1  Graph g = read_graph();
2  Tree mst;
3  Worklist wl = g.random_node();
4  foreach(Node n in wl) {
5    foreach(Neighbor nb of n) {
6      if(!nb.in_mst) {
7        mst.add_edge(n, m);
8        wl.add(nb);
9      }
10   }
11 }
```

Listing 3.7: Pseudocode of the spanning tree algorithm

processing.

The paralellism in this algorithm works inverse to Boruvka. As it starts with a single node, the initial stages of the algorithm are done sequentially. As more nodes are processed, eventually nodes outside the initial domain will be checked, thus allowing new parallel tasks to start participating in the processing.

3.5. Evaluation

All the algorithms required little work to be parallelized using our library. The main loops have been substituted with an invocation to the `parallel_domain_proc` algorithm template, and the only extra Lines are for initializing the `Domain` and

```
1  int sum = 0;
2  Graph::worklist wl;
3  Graph* g = readGraph();
4
5  for_each(g->begin_nodes(), g->end_nodes(), [&](Node* n) {
6    g->add_to_worklist(wl, n);
7  });
8
9  while(!wl.empty()) {
10   Node* current = *wl.front();
11   wl.pop_front();
12   if(!current) continue;
13
14   Node* lightest = findLightest(g, current);
15
16   if(lightest) {
17     sum += g->findEdge(current, lightest)->data();
18     g->add_to_worklist(wl, edgeContract(g, current, lightest));
19   }
20 }
21
22 return sum;
```

Listing 3.8: Serial version of Boruvka's algorithm

```
1  int sum = 0;
2  BGraph::worklist wl;
3  Graph* g = readGraph();
4
5  for_each(graph->begin_nodes(), graph->end_nodes(), [&](Node* n) {
6    graph->add_to_worklist(wl, n);
7  });
8
9  Domain2D<int, Node> plane(minx-1, miny-1, maxx+1, maxy+1);
10 parallel_domain_proc(graph, wl, plane,
11 [&](Graph* g,Node* current,Graph::worklist& wll,const Domain2D<int, Node>& d) {
12   Node* lightest = findLightest(g, current);
13
14   if(lightest) {
15     check_node_and_neighbors(g, d, lightest);
16     atomic_add(sum, g->findEdge(current, lightest)->data());
17     g->add_to_worklist(wll, edgeContract(g, current, lightest));
18   }
19 });
20
21 return sum;
```

Listing 3.9: Parallel version of Boruvka's algorithm

checking whether a node belongs to a subdomain. This is shown in Listing 3.9. This code computes the weight of the minimum spanning tree using Boruvka, and stores it in `sum`. This is an atomic integer, because all the tasks are accumulating in it the weight of the tree as they compute it. We used the C++11 lambda function notation to represent functions used as argument for algorithm templates, in Lines 5 and 11. The lambda functions used begin with the notation [&] to indicate that all the variables not in the list of arguments have been captured by reference, i.e., they can be modified inside the function. Line 5 is a for loop that initializes the worklist and stores it in `wl`. Then, Line 9 creates the domain, in this case with a two-dimensional plane that encompasses the full graph. Finally, the skeleton is run in Line 10. In Line 15, the helper method of the `Domain2D` class `check_node_and_neighbors` checks whether node `lightest` and all its neighbors fall within domain `d`. If not, it throws an out-of-domain exception.

In order to measure the complexity of the parallelization of the irregular algorithms using our library we resorted to the SLOC and cyclomatic number [91] metrics described in Section 1.5. Our measurements were performed considering the whole source code for each algorithm and version. The relative changes of these metrics are shown in Figure 3.4 as the percentual difference between the parallel and the sequential version. It can be seen that despite the irregularity of the algorithm, small changes are required in order to go from a sequential to a parallel version, and the growth of any complexity measure is at most 3% in the parallel version. In fact, in the case of the cyclomatic number, it is often lower for the parallel version than for the sequential one. This is because there are conditionals that are hidden by the library, such us the check for nonexistent workitems. This way, the simplicity of the parallelization of irregular algorithms using our library is outstanding.

The speed-ups achieved, calculated with respect to the serial version, are shown in Figure 3.6. The system used has 12 AMD Opteron cores at 2.2 GHz and 64 GB. The Intel icpc v12 with `-fast` optimization level was used. The inputs of the algorithms were:

Boruvka A graph defining an street map with $6 \cdot 10^6$ nodes and $15 \cdot 10^6$ edges, taken from the DIMACS shortest path competition [126]. In this graph, the nodes are labeled with the latitude and logitude of the cities, so we can use a two-dimensional domain.

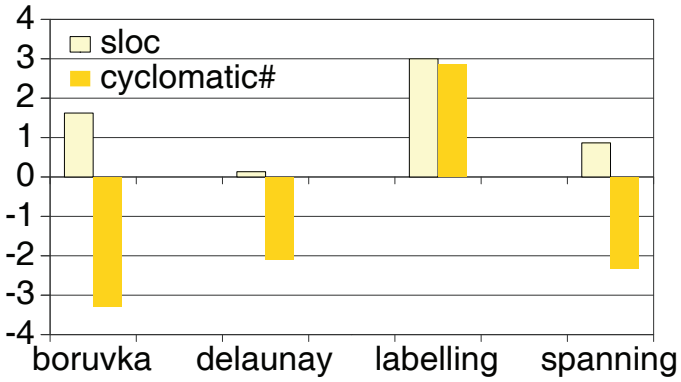

Figure 3.4: Relative percentages of the SLOCs and the cyclomatic number of the parallelized version with respect to the sequential one

Delaunay Mesh Refinement A mesh triangulated with Delaunay's triangulation algorithm with 10^5 triangles, taken from the Galois project input `massive.2` [79]. With this mesh, a graph is built where each node correspond to one triangle. We use the coordinates of the first vertex of the triangle as the label of the node, to use it with a two-dimensional domain.

Graph labeling Disjoint graph with $3 \cdot 10^6$ nodes and $8 \cdot 10^6$ edges distributed on at least 10^4 disconnected clusters, similar to those in [67]. In this graph, each node has a unique and consecutive ID in a one-dimensional domain.

Spanning tree A regular grid with 3000 height and 3000 width, where each node except the boundary nodes had 4 neighbors. The grid structure allows us to assign x and y coordinates to each node, therefore suiting it for a two-dimensional domain.

The parallel times were measured using the default behavior of generating two bottom-level subdomains per core used. Since the number of subdomains generated by our skeleton is a power of two, 32 subdomains were generated for the runs on 12 cores.

Figure 3.5 shows the running times of the experiments and the time of the sequential version used as baseline. The time with one thread is comparable to that of the sequential version, which shows the low overhead that our library introduces. This can be seen also in the speedups shown in Figure 3.6. This was expected because the irregular access patterns characteristic of these algorithms, coupled with the small number of computations in most of these benchmarks, turn memory bandwidth and latency into the main factor limiting their performance.

The speedups achieved are very dependent on the processing performed by each algorithm. Namely, labeling and spanning, which do not modify the graph structure, are the benchmarks that scale better. Let us remember that labeling only modifies data (the color of each node), while spanning inspects the graph from some starting point just adding a single edge to the output graph whenever a new node is found. Delaunay refinement operates on a neighborhood of the graph removing and adding several nodes and edges, but it also performs several computations. Finally Boruvka is intensive on graph modifications, as it involves minimal computations,

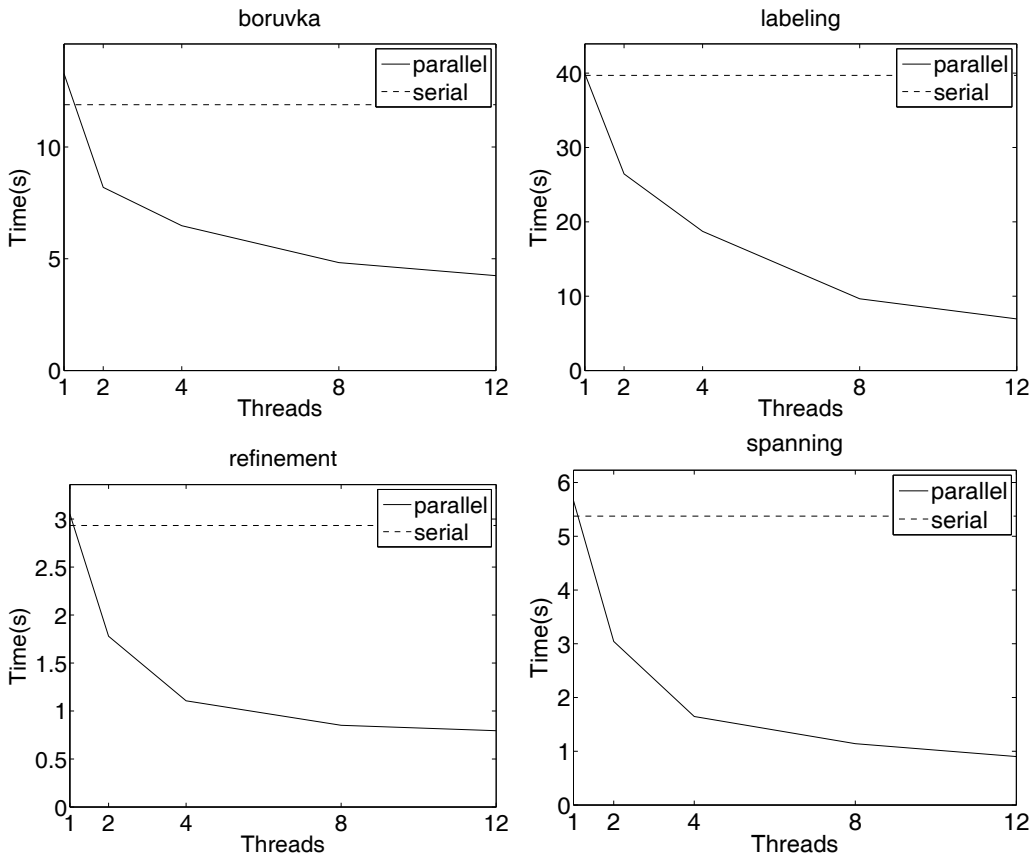

Figure 3.5: Running times for the benchmarks

and it removes and adds an enormous number of nodes and, particularly, edges. This way the latter two algorithms suffer from more contention due to synchronizations required for the simultaneous deletions and additions of their parallel tasks on the shared graph. An additional problem is that parallelization worsens the performance limitations of these algorithms due to the memory bandwidth because of the increasing number of cores simultaneously accessing the memory. For these reasons these are typical speedups for these applications [117][78].

The degree of domain over-decomposition can also affect the performance. Figure 3.7 shows the relative speedup achieved using 8 cores with several levels of over-decomposition with respect to the execution without over-decomposition, that is, the one that generates a single bottom-level subdomain per core. In the figure, n levels of over-decomposition imply 2^n subdomains per core. This way the results shown in Figure 3.6 correspond to the first bar, with one level of over-decomposition.

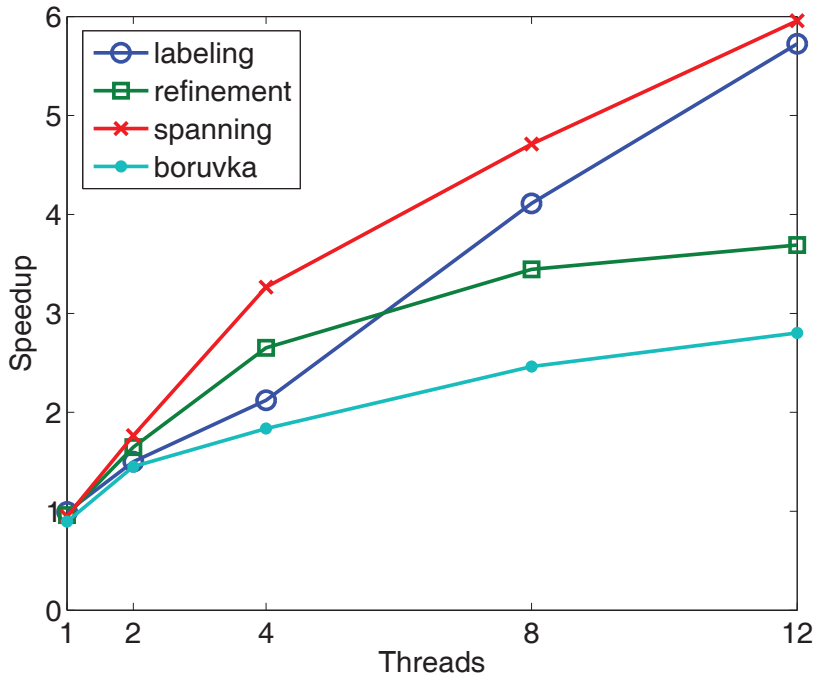

Figure 3.6: Speedups with respect to optimized serial versions

We can see that just by not over-decomposing the input domain, Delaunay refinement gets a very important performance boost, while spanning successfully exploits large levels of over-decomposition.

Figure 3.8 shows the percentage of elements that fall outside the domain, and therefore have to be deferred to upper levels of domain subdivision, also for runs with 8 cores. It is interesting to see that even when we are not using a small number of cores, and thus of subdivisions of the domain, the number of workitems aborted never exceeds 3% in the worst case. These values help us explain the results in Figure 3.7. Labeling has no conflicts because in its case the role of the domain is only to partition the tasks; when two tasks operate simultaneously on an area, the one with the smallest color will naturally prevail. So over-decomposition does not play any role with respect to conflicts in this algorithm; it only helps its load balancing. As for Delau- nay refinement, even when only 3% of its workitems result in conflicts, this ratio is proportionally much higher than for the other algorithms, and their individual cost is also larger. This way, although decreasing over-decomposition

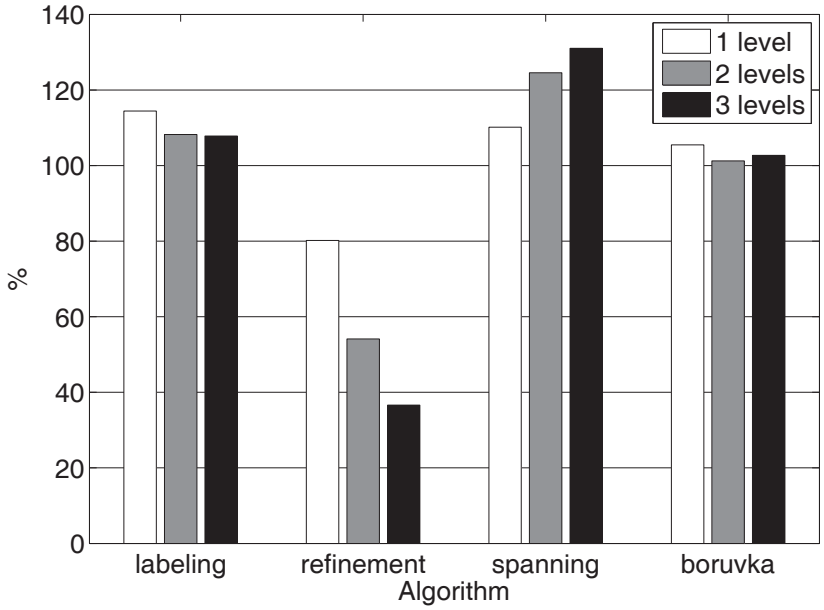

Figure 3.7: Relative speedup with respect to no over-decomposition in runs with 8 cores. 100 is the baseline, that is, achieving 100% of the speedup (i.e. the same speedup) obtained without overdecomposition.

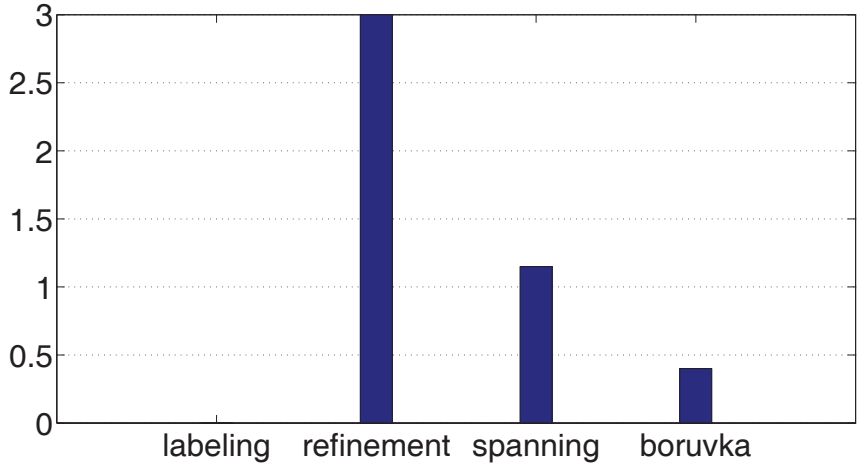

Figure 3.8: Percentage of out-of-domain elements running with 8 cores and 16 bottom-level subdomains

might reduce load balancing opportunities, this is completely offset by the important reduction in the number of conflicts. Spanning is the second algorithm in terms of conflicts, but two facts decrease their importance for this code. First, this algorithm begins with a single workitem from which the processing of neighboring domains are later spawned. This way if there is no over-decomposition some threads begin to work when the processing reaches their domains, and stops when their domain is completely processed. This leads to a very poor usage of the threads. Over-decomposing allows threads that finish with a given subdomain to begin working on new domains reached by the processing. The second fact is that delayed workitems because of conflicts often find that they require no additional processing when they are reconsidered in an upper level of subdivision because they were already connected to the spanning tree by their owner task at the bottom level. Finally, Boruvka has relatively few conflicts and their processing cost is neither negligible nor as large as in Delaunay refinement. Thus, a small degree of over- decomposition is the best in terms of balancing the amount of work among the threads, potentially more so for an increasing number of subdomains, and the number of conflicts, which also increase with the number of subdomains.

3.6. Exploring the Configuration Capabilities of the Skeleton

Skeletons often provide either no tools or very restrictive possibilities to control essential aspects of their execution. This can result in suboptimal performance because parallel applications are highly sensitive to implementation decisions such as the work partitioning algorithm, the degree of work decomposition, or the data structures used. For example, [37] describes some disadvantages of the highly automated template implementation of skeletons, such as taking wrong decisions that the users cannot fix.

The reduced parametrization capability of a skeleton is a small problem and can be in fact very justified in regular algorithms, as it is easier or even straightforward to derive heuristics to choose good parallelization strategies for them. This is the case of skeletons for purely data parallel operations, which can offer only

block distributions [30][120][38] or even totally hide the work decomposition from the user [89][45]. Nevertheless, this is not a good approach for irregular applications, and in particular for the amorphous data-parallel ones, where the patterns of computation can widely vary between algorithms, and the best work decomposition for a given algorithm can follow different strategies for different inputs, there being besides a large variety of partitioning strategies [106]. Therefore in these applications it is critical that users can experiment with several possibilities, and using a high-level approach such as a skeleton should not preclude but facilitate this. In this regard, although there are skeletons [112][83][52] that allow total control of the task decomposition, they require users to programmatically specify all the details of this decomposition except for basic trivial cases like ranges, and they do not support amorphous data-parallelism. Since we are convinced that convenient and flexible parameterization is key to the success of skeletons in this new area, several practical mechanisms to control the execution of `parallel_domain_proc` were designed and implemented during the development of this thesis. Thanks to them, our strategy to parallelize irregular applications allows users to explore a large space of possible concrete implementations without renouncing to the high level structured approach and good programmability enabled by the skeleton. This section describes and evaluates these mechanisms, showing their large impact on performance, and thus the relevance of enabling the user to (easily) control them.

3.6.1. Skeleton Behavior Configuration

A first configurable element is the boolean template parameter `redirect` mentioned in Section 3.3.5, which controls the behavior of tasks that work at the bottom level of decomposition when they find an element outside their subdomain in their work list. Since the initial work lists in bottom level tasks only contain elements within their assigned subdomain, this only happens in algorithms whose processing can generate new workitems. When `redirect` is `false`, which is its default, the usual policy of delaying the processing of the workitem to later tasks that will be run with larger subdomains is applied. When it is `true`, if the task associated to the workitem subdomain at the bottom level of subdivision or a parent of it is already running, the usual policy is also followed. Otherwise, the workitem is placed in the work list of the owner bottom level task, and the task is submitted to execution. In

order to enable this redirection process, the skeleton does not run tasks whose work lists are empty. While this flag controls a very specific situation, it is critical to enable the parallelization of the large class of algorithms that begin with a single node in the graph from which the processing propagates throughout the whole structure.

While `redirect` is required to parallelize certain algorithms, there are other implementation variations that are design decisions that can lead to different performance for different algorithms and inputs. In this thesis we will consider three implementation variations of this kind. The first one is the usage of different domain decomposition algorithms, which can lead to different load balance and number of conflicts among the tasks. The `parallel_domain_proc` skeleton allows to explore this possibility by writing domains with user-defined splitting strategies or simply by using or deriving from domain classes provided by the library that implement predefined partitioning schemes. The ones we have developed are:

- **Clustered Domain (cd)** tries to assign nodes that are close in terms of edges between them to the same subdomain. This is achieved by means of a clustering process in which each cluster starts from a random node and expands from it following a breadth first search. The clustering process is performed only once, generating all the bottom level domains required, and it stops when all the nodes have been assigned to subdomains. The intermediate levels of domain decomposition are generated by aggregating neighbor clusters following a bottom-up process. The fact that the decomposition does not follow the top-down approach explained in Section 3.2 is hidden inside the domain and it is totally oblivious to the user and the skeleton, which uses the same interface and logical process as for any other domain decomposition scheme.
- **Clustered Tree Domain (ctd)** Instead of building all the subdomains simultaneously, this alternative starts with one source node. It does a breath-first search and adds nodes to a subdomain, until half of the nodes are in it. This splits the whole graph in two subdomains with almost the same number of elements. Then, it choses one source node from each subdomain, and it repeats the subdivision process. This continues recursively until the number of desired subdomains is reached. This generally creates subdomains with a similar number of nodes, providing better work balance.

- **DomainND (d2d)** Very often the domains are N-dimensional spaces in which each dimension is associated to one of the data items of the graph elements and its extent is given by the range defined by the minimum and the maximum value found in the graph elements for that item. This domain is easily divisible by cyclically splitting each one of its dimensions (i.e., the i-th subdivision splits the domain(s) across dimension $i \bmod N$) until the required number of subdomains are generated. This was the only scheme tested in Section 3.5.

Another possibility is changing the number of levels of decomposition of the domains. Generating a bottom level domain, and thus a bottom level parallel task, per core available is a reasonable option. However, since the runtime of our skeleton is built on top of Intel TBB [112] and efficiently exploits task-stealing, it can provide improved load balancing if the domain is over-decomposed, so that the task-stealing scheduler can profit from the excess parallelism created. We have simplified the exploration of this possibility by adding a new optional parameter to the constructor of the domains, `overdecomposition`. This parameter requests their decomposition in $2^{\texttt{i}}$ subdomains per core, the default being `overdecomposition` $= 0$, that is, a bottom level subdomain per core.

Finally, we can experiment with different data structures. For example, work lists, which are dynamic structures from which elements are being continuously removed, but which in some algorithms also dynamically receive new workitems, can play a crucial role in performance. Given these characteristics, it looks like regular (STL) lists are a good alternative for them, as they perfectly model the behavior required, and they where in fact the work lists used in 3.5. Other implementations can be however considered. This way we have also built a (STL) vector-based work list that pushes new workitems at the end of the vector and which never removes the already processed workitems. Rather, it simply increases an internal pointer that indicates the first unprocessed workitem in the vector. This way this implementation trades space inefficiency for better locality and reduced allocation and deallocation cost, as vectors grow by chunks.

The programming effort required to explore these configuration variations is minimal. The `redirect` flag simply requires providing a boolean, while the level of domain decomposition is specified with a single method invocation to the domain object. Finally, the domains, work list containers, context objects, etc. are template

classes, so they can accommodate any classes either provided by the library of built by the user for any of the objects involved. Of course this includes the skeleton, which is a function template that automatically adapts to the types of its arguments. This way exploring the possibilities available only requires changing the type of the associated object.

3.6.2. Evaluation

In addition to the algorithms we described in Section 3.5, we implemented two additional benchmarks to test the programmability of the library. These algorithms help to visualize the difference between the different configurations of the skeleton.

- **Independent Set** (IS) computes a maximal independent set of a graph, which is a set of nodes such that (1) no two nodes share the same edge and (2) it cannot be extended with another node. This greedy algorithm labels each node with a flag that may be in one of three states: Unmatched, Matched and NeighborMatched. All the flags begin in the Unmatched state. An unmatched node is selected from the graph. If none of its neighbors are matched, then the flag for the node is set to matched and all of its neighbors flags are set to NeighborMatched. This process continues until there are no more unmatched nodes, in which case, the nodes with matched flags are a maximal independent set.

- **Single-Source Shortest Path** (SSSP) solves the single-source shortest path problem with non-negative edge weights using a demand-driven modification of the Bellman-Ford algorithm. Each node maintains an estimate of its shortest distance from the source. Initially, this value is infinity for all nodes except for the source, whose distance is 0. The algorithm proceeds by iteratively updating distance estimates starting from the source and maintaining a work list of nodes whose distances have changed and thus may cause other distances to be updated.

The experiments were performed in a system with 2 Intel Xeon E5-2660 Sandy Bridge-EP CPUs (8 cores/CPU) at 2.2 GHz and 64 GB of RAM, using g++ 4.7.2

with optimization flag `-O3`. The graph, node and edge classes in these experiments used were taken from the Galois system [79], as they were found to be more efficient than the locally developed ones used in 3.5 and the skeleton transparently supports any classes. The inputs were a road map of the USA with 24 million nodes and 58 million edges for Boruvka, IS and ST, a road map of New York City with 264 thousand nodes and 733 thousand edges for SSSP –both maps taken from [126]– and a mesh with 1 million triangles taken from the Galois project for DMR. Since Spanning Tree and Single-Source Shortest Path begin their operation with a single node from which the computation spreads to the whole graph, their parallelization has been performed activating the `redicted` optional feature of `parallel_domain_proc`, which is not used in the other benchmarks.

Table 3.1 shows the baseline times for the experiments, obtained from pure sequential implementations. They are neither based on our skeleton nor on any other parallelization mechanism that could add any overhead, and they use the best data structures for graphs and work lists, which are vectors, as we will soon see. Figure 3.9 shows the speedups obtained with respect to a sequential execution for each benchmark using different number of cores. We tried six combinations based on the usage of the three domain decomposition strategies described in Section 3.6.1 (cd for Clustered Domain, ctd for Clustered Tree Domain and d2d for the DomainND in two dimensions) and two work list containers, namely standard (std::)lists (l) and vectors (v), both using the default C++ allocators. The executions followed the default policy of generating one bottom level task per core. The slowdown for a single core gives an idea of the overhead of the skeleton, which can be up to three slower than the sequential version in these runs.

Vector-based work lists clearly perform better than list-based ones despite being more memory greedy, as they do not remove already processed elements (see de-

Table 3.1: Baseline times for the algorithms

Benchmark	Serial time (s)
Boruvka	17.95
IS	0.34
DMR	13.66
ST	1.61
SSSP	0.63

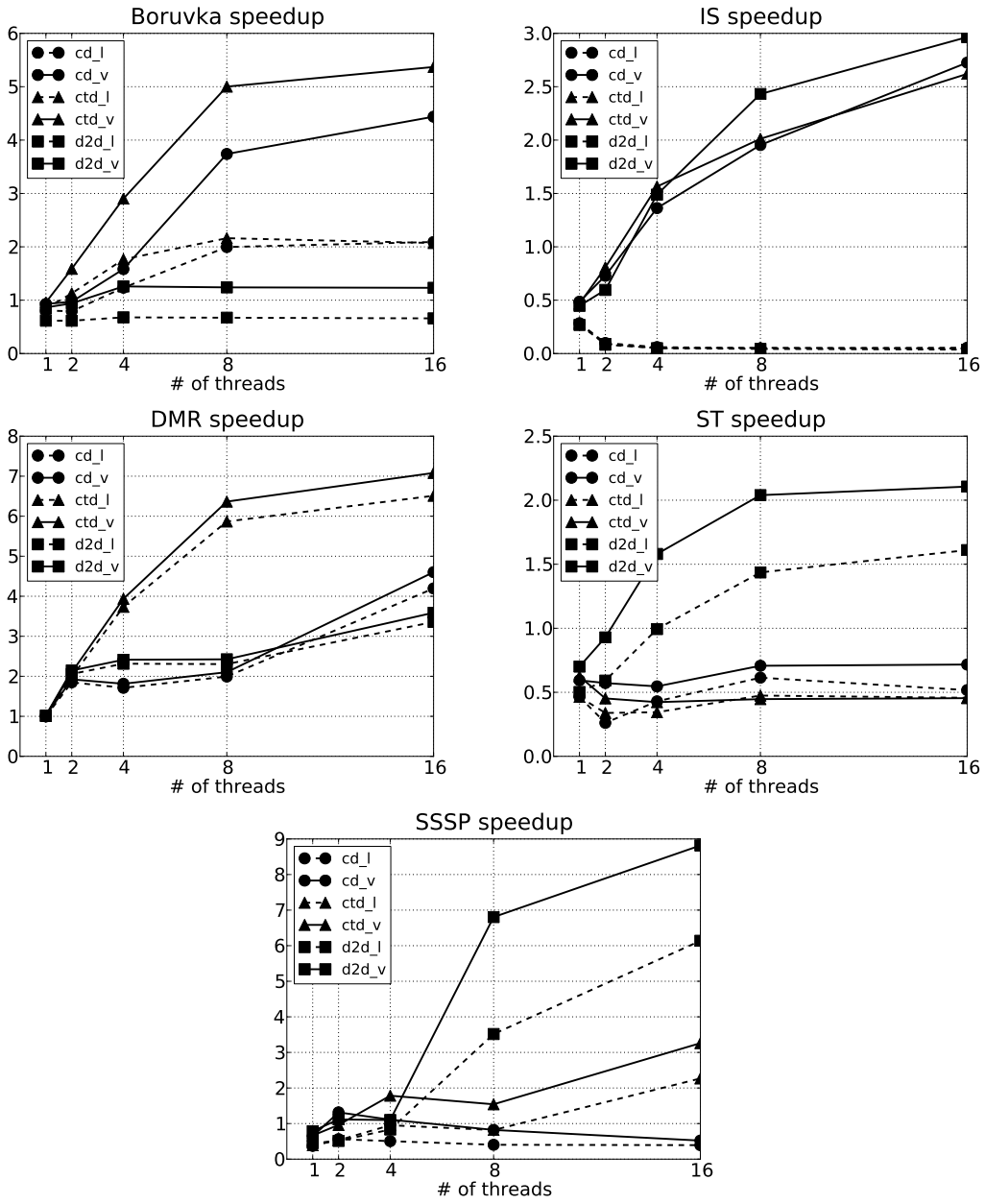

Figure 3.9: Speedups using different domains and containers

scription in Section 3.6.1). Thanks to the reduced memory management cost and better locality, the traversal of the worklist is much more efficient when using vectors than when using lists. While in some algorithms the extra cost of lists is relatively small (DMR, or ST for some domain partitionings), and lists are in fact negligibly

faster in ST with domain ctd for 8 and 16 cores, in others the disadvantage of lists is enormous. The best example is IS, where the versions that use lists obtain much worse speedup than those with vectors.

We suspected that the most important reason for the bad performance of lists is their requirement to continuously allocate and deallocate items. This operation is even more expensive in a multithreaded program, where the memory management provided by the C++ runtime is thread-safe, with the associated synchronization costs. In order to prove this, we wrote a customized allocator class that acts as a pool, thus minimizing the number of invocations to the underlying thread-safe memory manager. Our allocator has a thread-safe and a faster non-threadsafe version. We could use the last one thanks to the fact that each worklist is always accessed by a single thread of the skeleton. The results are shown in Figure 3.10, where lists use our allocator; a change that was straightforward thanks to the easy configurability of our skeleton, just requiring modifications in a couple of lines. Using our pool allocator greatly reduces the gap in performance between vectors and lists, up to the point of almost achieving the same speedups with both containers and in some cases, like ST and SSSP, improving them. The most significant case is IS, which did not show any speedup when using lists with the standard allocator (in Figure 3.9) but now presents a reasonable scalability with the list container. Figure 3.11 shows the speedup that the list-based codes obtain when using our allocator with respect to the original experiments using the standard one in the executions using 16 cores. While IS achieves very large speedups of 32.4, 35.1 and 59.8 when using the domains cd, ctd and d2d, respectively, the other benchmarks become between 1.07 and 2.51 faster with our allocator, the mean speedup for them being a still noticeable 1.53. If IS is also taken into account, the average speedup is 9.7.

The type of domain decomposition also plays a critical role in performance, there being not a clear winner. The DomainND strategy is usually the best one for IS, ST and SSSP, while the Clustered Tree Domain offers the best performance for Boruvka and DMR. The need to allow programmers to easily test different configurations of the skeleton execution is further supported by the fact that while in some applications a decomposition algorithm is always the best across the board, this is not the case in others. For example, while for 8 and 16 cores SSSP achieves the best performance with DomainND, the best speedups for 2 and 4 cores are achieved with the

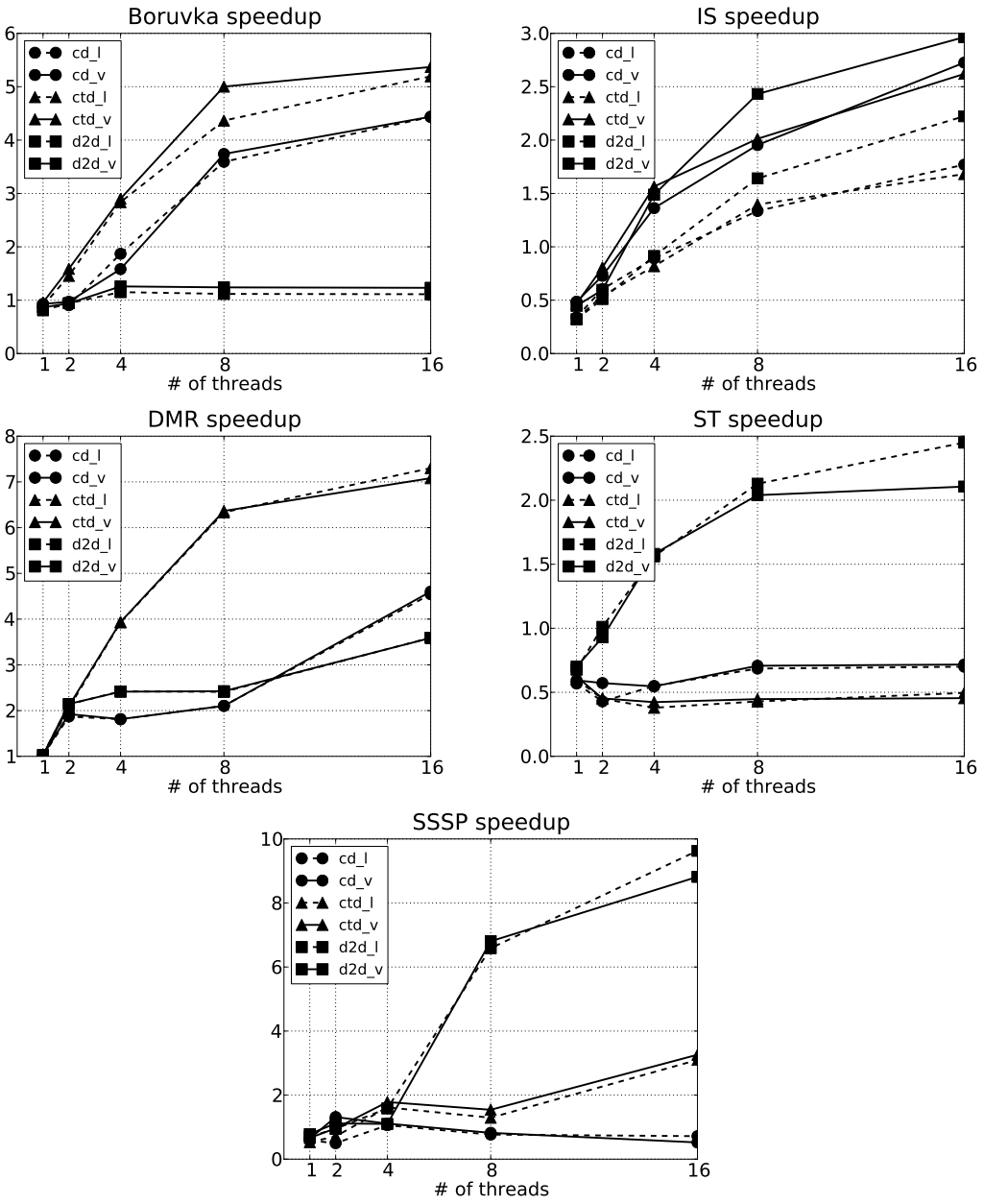

Figure 3.10: Speedups using different domains and containers. In this figure, lists use our pool allocator

Clustered Domain and the Clustered Tree Domain strategies, respectively. Similary, while DomainND is also the best strategy for IS for runs with 8 and 16 cores, it is the worst partitioning when we only have 2 cores. Also, in some specific situations

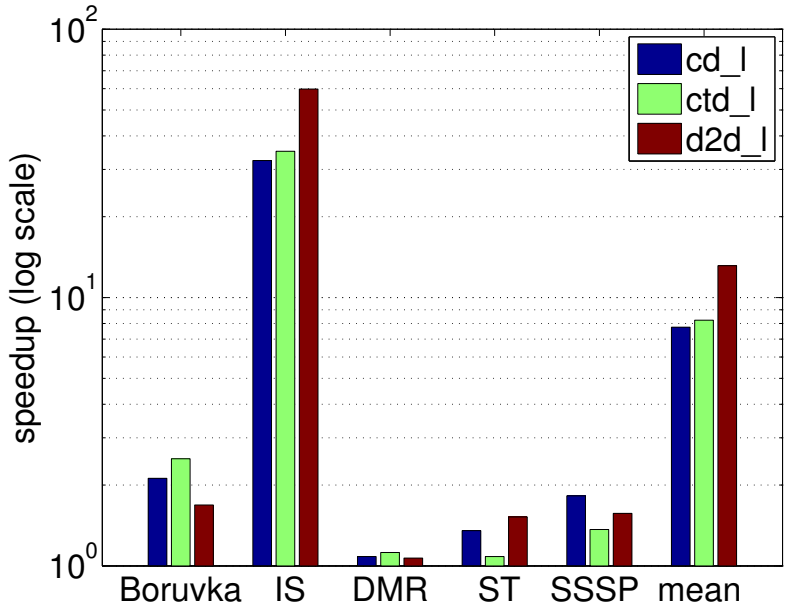

Figure 3.11: Speedups in the experiments with lists using the pool allocator with respect to the lists using the standard allocator in the runs with 16 cores

all the partitioning algorithms can provide a very similar speedup. This is the case of DMR with two threads. This algorithm performs a lot of work per workitem, so it tends to scale linearly if the domains are balanced in terms of work, and the number of conflicts due to neighborhoods that extend outside the local domain are minimized. When only two domains are used, the number of conflicts is minimal due to the small number of subdomains. If in addition, the domains are built using reasonable heuristics like the ones considered in this section, they will probably be reasonably balanced. Both circumstances favor the behavior observed. Another reason for the complex behavior of these applications in terms of performance is that in many of them the amount of work per workitem is highly variable, and sometimes impossible to predict in advance. This is the case of the DMR bad triangle cavities, whose extent can only be known when they are explored, and where the number of new bad triangles generated can only be known after the re- triangulation. Another example is Boruvka, whose amount of work per node is proportional to the number of edges to be contracted. This number not only depends on the initial number of edges of each node, but also on the sequence of nodes contracted before the one considered.

All in all, the best decomposition strategy depends on the application, the number of cores, and the kind of input graph, as it can favor a specific partitioning strategy [106]. Given the complexity of the subject, it is difficult to make a priori selections of the domain decomposition algorithm, and although the generic algorithms we propose can obtain good results, a better understanding of the application can allow users to create domains that can obtain better results.

The impact of over-decomposition on performance is analyzed in Figure 3.12 with experiments using 16 cores. It shows the relative speedup of each one of the six possibilities tested in Figure 3.9 with respect to their own execution with no over-decomposition, that is, one in which one bottom level task is created per core. As we explained in Section 3.6.1, a level of decomposition `i` means generating $2^{\texttt{i}}$ tasks per core, thus for $\texttt{i} = 0$ the speedup in the figure is always 1. The -1 level, which generates 8 tasks, was tried to test if the lower number of task merges could improve the performance, which happened very seldom. Over-decomposition, which is very easy to apply with our skeleton, can largely improve performance, even when we consider the choices that achieved the best performance in Figure 3.9. This way, d2d_v, which was the best strategy for 16 cores for IS, ST and SSSP, further increases its performance by 10%, 30% and 50%, respectively, when 2 tasks per core are generated.

Overall the skeleton achieves performance similar to that found in the bibliography for manually-tuned parallel implementations of these applications. This is the case for example for DMR in [117], although this is only a qualitative observation given the different hardware and inputs tested. Regarding the absolute speedups achieved, we must note that the performance of this kind of applications is more limited by memory latency and bandwidth than that of applications with regular access patterns and more CPU operations per input data item.

3.7. Related work

Since our strategy relies on partitioning the initial work to perform in chunks that can be mostly processed in parallel, our approach is related to the divide-and-conquer skeleton implemented in several libraries [34][112][30][52]. Nevertheless, all

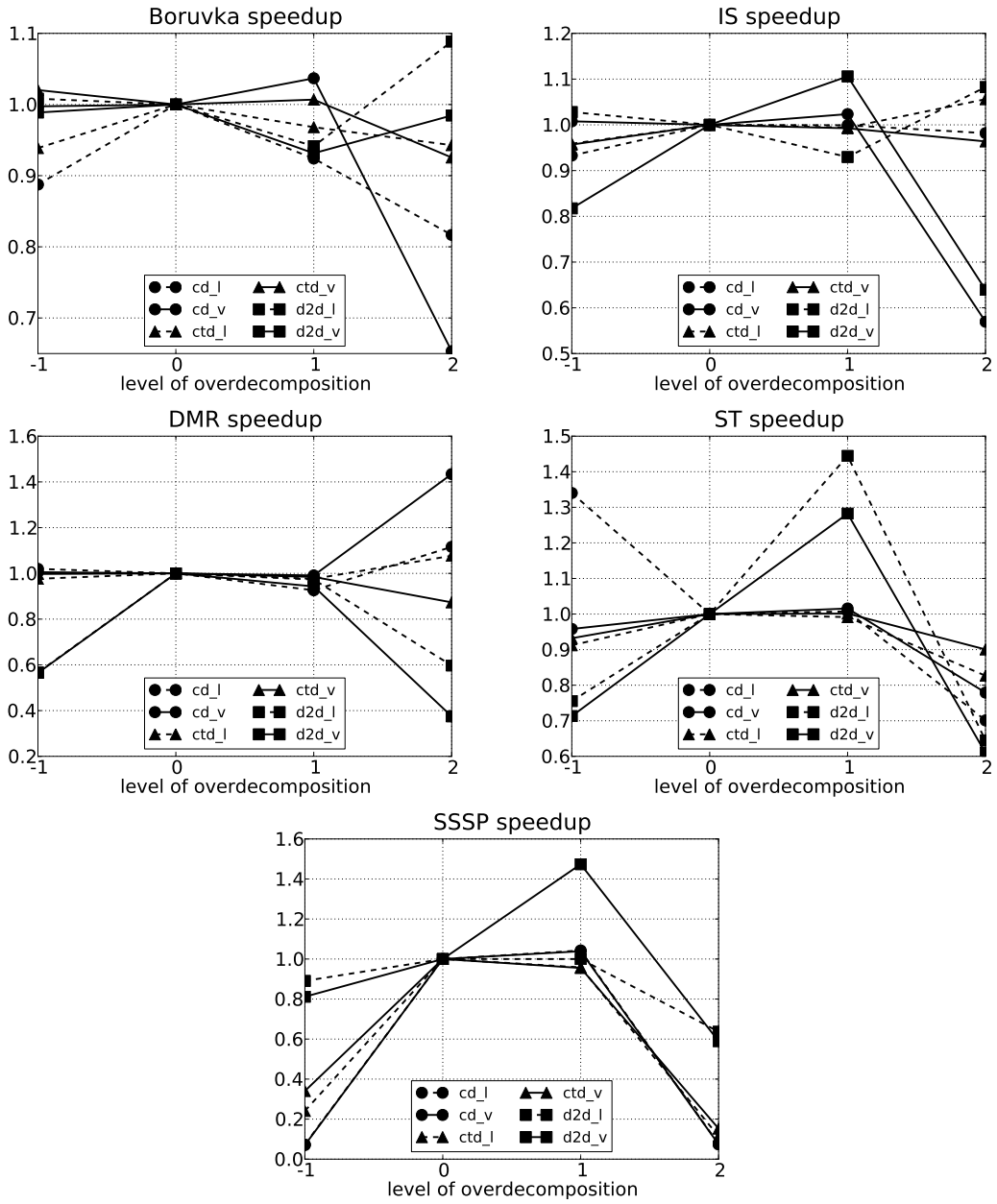

Figure 3.12: Speedups using different levels of decomposition with respect to no over-decomposition in runs with 16 cores

the previous works of this kind we are aware of are oriented to regular problems. As a result those skeletons assume that the tasks generated are perfectly parallel, providing no mechanisms to detect conflicts or to deal with them once found. Neither

do they support the dynamic generation of new items to be processed by the user provided tasks. This way, they are not well suited to deal with the irregular problems we are considering.

One of the approaches to deal with amorphous data parallel algorithms is Hardware or Software Transactional Memory (HTM/STM) [69]. HTM limits, sometimes heavily, the maximum transaction size because of the hardware resources it relies on. The Blue Gene/Q was the first system to incorporate it, and although it is present in some Top500 supercomputers, its adoption is not widely spread. Several implementations exist for STM [65][116], but their performance is often not satisfactory [23]. With STM, the operations on an irregular data structure are done inside transactions, so when a conflict is detected, as overlapping neighborhoods for two nodes, it can be rolled back.

Another option is Thread Level Speculation (TLS), which from a sequential code creates several parallel threads, and enforces the fulfillment of the semantics of the source code using hardware support [64][32] or software methods [111][31][61][4]. But, just as the solutions based on transactional memory, TLS cannot take advantage of the knowledge about the data structure and the algorithm nature as ours does.

The Galois system [79] is a framework for this kind of algorithm that relies on user annotations that describe the properties of the operations. Its interface can be simplified though, if only cautious and unordered algorithms are considered. Galois has been enhanced with abstract domains [78], defined as a set of abstract processors optionally related to some topology, in contrast to our concept of set of values for a property of the items to process. Also, these domains are only an abstraction to distribute work, as opposed to our approach, where domains are the fundamental abstraction to distribute work, schedule tasks and detect conflicts, thus eliminating the need of locks and busy waits found in [78]. Neither do we need over-decomposition to provide enough parallelism, which allows for higher performance in algorithms with costly conflicts, as Delaunay refinement shows in Figure 3.7. Finally, lock-based management leads conflicting operations in [78] to be repeatedly killed and retried until they get the locks of all the abstract processors they need. Nevertheless, the computations that extend outside the current domain in our system are just delayed to be retried with a larger subdomain. This way the number of attempts of a con-

flicting task is at most the number of levels of subdivision of the original domain. With the cautions that the input and implementation languages are not the same and that they stop at 4 cores, our library and Galois yield similar speedups for Delaunay in a comparable system [78].

Chorus [85] defines an approach for the parallelization of irregular applications based on object assemblies, which are dynamically defined local regions of shared data structures equipped with a short-lived, speculative thread of control. Chorus follows a bottom-up strategy that starts with individual elements, merging and splitting assemblies as needed. These assemblies have no relation to property domains and their evolution, i.e., when and with whom to merge or split, must be programmatically specified by the user. We use a top-down process based on an abstract property, and only a way to subdivide its domain and to check the ownership are needed. Also, the evolution of the domains is automated by our library and it is oblivious to the algorithm code. Moreover, Chorus is implemented as a language, while we propose a regular library in a widely used language, which eases the learning curve and enhances code reusability. Also, opposite to Chorus' strategy, ours does not require locks, which favors scalability, and there are no idle processes, so the need for over-decomposition is reduced. Finally, and in part due to these differences, our approach performs noticeably better on the two applications tested in [85].

Partitioning has also been applied to an irregular application in [117]. Their partitioned code is manually written and it is specifically developed and tuned for the single application they study, Delaunay mesh generation. Additionally, their implementation uses transactional memory for synchronizations.

Finally, the concept of hierarchical partitioning has also been estudied, for example in [93], as a means to improve data locality through several processing elements, and in the memory hierarchy of each processing element. But their solution is only applicable to regular algorithms and data structures, and as in the case of the Galois system, their domain is just an abstraction of the number of processing elements; unlike ours, it is not defined from the properties of such data and it is not configurable.

3.8. Conclusions

Amorphous data parallelism, found in algorithms that work on irregular data structures is much harder to exploit than the parallelism in regular codes. There are also few studies that try to bring structure and common concepts that ease the parallelization of these algorithms. In this chapter we explore the concept of domain on the data to process as a way to partition work and avoid synchronization problems. In particular, our proposal relies on (1) domain subdivision as a way to partition work among tasks, on (2) domain membership, as a mechanism to avoid synchronization problems between tasks, and on (3) domain merging to join worksets of items whose processing failed within a given subdomain, in order to attempt their processing in the context of a larger domain.

An implementation of our approach based on a skeleton operation and a few classes with minimal interface requirements is also presented. An evaluation using several benchmarks indicates that our algorithm template allows to parallelize irregular problems with little programmer effort, providing speed-ups similar to those typically seen for these applications in the bibliography.

Also, we have extended and evaluated the configurability of our skeleton for amorphous data-parallel applications. These applications offer a large number of implementation possibilities based on the use of different data structures, levels of work decomposition and work partitioning algorithms, which are richer than those of regular algorithms. The ability to easily experiment with these possibilities is very important for irregular applications because deriving heuristics to decide the best configuration for each given algorithm, input and computer to use is much more difficult than in the case of regular applications, or even impossible.

Our experience has shown that many alternative configurations can be explored with this skeleton with very little effort. We have also seen that the impact on performance of each implementation decision, even taken isolatedly, can be enormous and that the best alternative depends on the algorithm and the number of cores available. Also, although the generic options provided in the library provide reasonable performance, users can define and use their own decomposition algorithms, data structures, etc. specifically targeted to their particular problem to achieve better performance.

Chapter 4

Library for task parallelism with detection of dependencies

One of the outstanding difficulties of the development of parallel applications is the synchronization of the different units of execution. Different parallel programming models and tools provide different synchronization techniques, which can be implicitly or explicitly invoked by the user. The data-parallel paradigm [70, 24, 48], for example, provides implicit synchronization points after each pair of parallel computation and assignment, being this one of the reasons why this paradigm is one of the most effective in terms of intuitiveness for the programmer. Unfortunately, the patterns of parallelism supported by this paradigm are too restrictive for many parallel applications. It is also the case that applications that greatly benefit from data-parallelism can further increase their performance when they are enriched with the possibility of exploiting more ambitious out-of-order scheduling [25, 27, 26] for the tasks in which their computations can be decomposed. Applications that benefit from complex patterns of parallelization and scheduling have usually to resort to paradigms and tools where explicit synchronization mechanisms are available in order to ensure that tasks satisfy their dependencies [22, 112, 99]. The implied required study of the dependencies among the tasks, and the subsequent specification of the corresponding synchronizations, result in an increased programming complexity, which is the cost to pay for the flexibility in the patterns of parallelism and dependencies among such tasks.

In this chapter we introduce a practical library-based approach to enable the expression of arbitrary patterns of parallel computation while avoiding explicit synchronizations. Our proposal is almost as flexible as explicit synchronization but without the complexity it brings to parallel programming. It also frees programmers from having to explicitly indicate which are the dependencies for their tasks. Our proposal, which has been implemented in the widely used C++ language, relies on the usage of functions to express the parallel tasks, the dependencies among the functions being solely provided by their arguments. This way, an analysis of the type of the arguments of a function whose execution has been requested, coupled with a comparison of these arguments with those of the functions that have already been submitted to execution, suffices to enforce the dependencies among them.

4.1. DepSpawn: An argument-based synchronization approach

As we have anticipated, our proposal is based on expressing the parallel tasks as functions that only communicate through their arguments. The types and memory positions of the arguments of the functions that define each parallel task are analyzed by our framework, called DepSpawn, to detect and enforce the dependencies among those tasks. Our library has three main components:

- The kernel is the function `spawn`, which requests a function to be run as a parallel task once all the dependencies on its arguments are fulfilled.
- The template class `Array`, which allows to conveniently express a dependency on a whole array or a portion of it.
- A few explicit synchronization functions to wait for the completion of all the pending tasks, to wait only for those that access some specific arguments, and to allow a task to early release dependencies.

We now describe these components in turn, followed by a small discussion on the implementation of the library.

4.1.1. Spawning parallel tasks

Function `spawn` accepts as first argument the name of the function whose execution as a parallel task is requested, followed by the comma-separated list of the arguments to the function. Since all the communications between parallel tasks must take place through their arguments, the function should have return type `void`, that is, act as a procedure. Functions returning a value are also accepted by `spawn`, but the returned value will be lost, as `spawn` does not return any value. It is worth mentioning that our library does not make any assumption regarding global variables. Thus if the user wants to track dependencies on these variables, she should pass them as parameters to the corresponding functions. The type of the corresponding function parameter will indicate whether the variable will only be read or can be also modified.

Table 4.1 summarizes the interpretation that `spawn` makes of each parameter or formal argument of a function depending on its type. Arguments passed by value are necessarily only inputs to the function, since any change made on them in the function is actually performed on a local copy. As for arguments passed by reference, which are marked in C++ by preceding the parameter name by a '`&`' symbol, the situation depends on whether the reference is constant or not. Constant references, which are labeled with the `const` modifier, do not allow to modify the argument, so they can only be inputs. Non-constant references however can modify the argument, therefore they are regarded as inputs and outputs. Table 4.1 reflects that pointers are regarded as inputs or inputs and outputs following the same rules as any other data type, and that the dependency is defined on them, not on the memory region they point to. This behavior has been chosen for two reasons. First, it is consistent with

Table 4.1: Behavior of `spawn` for each parameter type. `A` is any arbitrary data type. Modifiers in brackets do not change `spawn`'s behavior.

Type	Short description	Interpretation of `arg`
`[const] A arg`	Argument by value	input
`A& arg`	Argument by reference	input and output
`const A& arg`	Argument by constant reference	input
`[const] A* arg`	Pointer by value	input (not `*arg`)
`[const] A*& arg`	Pointer by reference	input and output (not `*arg`)
`[const] A* const & arg`	Pointer by constant reference	input (not `*arg`)

the treatment of the other data types. Second, the library cannot make reasonings on the dependencies generated through the usage of a pointer, as it is impossible to know the extent of the memory region that will be accessed through a pointer from its declaration.

Given the aforementioned interpretation associated to each function argument, and using the information on its address and length, `spawn` learns the data dependencies between the tasks submitted to parallel execution. It then guarantees that each task is run only once all its dependencies with respect to the previously spawned tasks are satisfied. In this regard, it must be outlined that our library releases the dependencies of a task only when the task itself and all its descendants (i.e. the tasks directly or indirectly spawned from it) finish. This way the release of a dependency by a task implies that all what would have been part of the serial execution of the task has been executed.

In order to formally detail the semantics of the programming model provided by our library, we make the following definitions:

- We say that a task T is *requested* when, during the execution of the program, the corresponding `spawn` invocation is made. A requested task can be waiting for its dependencies to be fulfilled, executing, or finished.
- A task T is an *ancestor* of task U if T requests U or, recursively extending this definition, it requests an ancestor of U.
- A task T is a *descendant* of a task U if U is an ancestor of T.
- We call *sequential execution* of the program the one that takes place if the spawn requests are replaced with standard function calls.
- A task T *logically precedes* task U if during the sequential execution of the program T is executed before U.
- A task T *dynamically precedes* task U if during the execution of the program T is requested before U (by the same or another thread of execution).
- A task T is said to have a *dependency* with a preceding task U if there is at least one memory position in common in their arguments that at least one of them could modify.

In our framework a requested task T waits for the completion of a task U if

1. U dynamically precedes T, U is not an ancestor of T, and T has a dependency with U; or

2. U is a descendant of a task that fulfills condition (1.).

Condition (2) is derived from the fact that in our framework a task does not release its dependencies until all its children finish their execution. Let us now analyze the implications of these conditions. Condition (1) indicates that a new task respects all the dependencies with the tasks that were requested before during the dynamic execution of the program, excluding of course its ancestors. In order to provide a formal definition of the guarantee provided by this condition, we will name tasks according to their order of execution in the sequential execution of the program. This way, we will call T_i the i-th task initiated during the sequential execution of the program that is not nested inside another task, T_{ij} will be the j-th task initiated by task T_i that is only nested in task T_i, and so on. Using this nomenclature, condition (1.) only guarantees that task $T = T_{x_0 x_1 \ldots x_n}$ complies with the dependencies with the preceding tasks $P_T = \{T_{x_0 x_1 \ldots x_{i-1} j}\,,\ \forall 0 \leq i \leq n, 1 \leq j < x_i\}$. The reason is that these are the only tasks that we know for sure that are requested before T in any execution. Among them, the focus in on the subset of those that actually generate dependencies with T, defined as $DP_T = \{U \in P_T \ / \ T$ has a dependency with $U\}$.

Additionally, condition (2.) guarantees that T is executed after the tasks that have an ancestor in DP_T, that is, Descendants(DP_T), where Descendants(S) = $\{T_{x_0 x_1 \ldots x_m x_{m+1} \ldots x_n} \,/\, T_{x_0 x_1 \ldots x_m} \in S, n > m\}$. Notice that both the tasks in DP_T and Descendants(DP_T) necessarilyl ogically precede T.

Figure 4.1, in which time runs from left to right and the spawn of new tasks is represented as a vertical line that leads to a new horizontal line of execution, helps illustrate these ideas. The tasks are labeled using the naming just defined, and it shows a concrete temporization in a parallel execution of an application. Following our definition, $P_{T_{42}} = \{T_1, T_2, T_3, T_{41}\}$ for task T_{42}, as these are the tasks that are not ancestors of T_{42} that will be requested before it no matter which is the exact length or temporization of execution of the different tasks. In this figure we assume

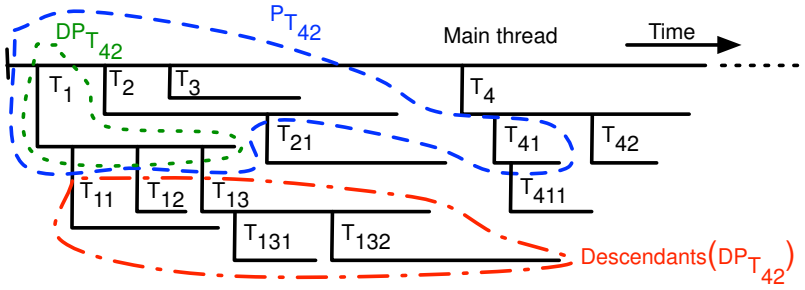

Figure 4.1: Preceding tasks $P_{T_{42}} = \{T_1, T_2, T_3, T_{41}\}$ for task T_{42}. Assuming that out of them only T_1 generates dependencies with T_{42} (i.e. $DP_{T_{42}} = \{T_1\}$), Descendants$(DP_{T_{42}}) = \{T_{11}, T_{12}, T_{13}, T_{131}, T_{132}\}$ is also depicted.

that out of these four tasks, T_{42} only has dependencies with respect to T_1, thus $DP_{T_{42}} = \{T_1\}$. Finally, the figure also depicts Descendants$(DP_{T_{42}})$, which are all the descendants of T_1. According to the rules just described, T_{42} will not start running until T_1 and all its descendants finish. Nevertheless, since T_2, T_3 and T_{41} are guaranteed to be analyzed before T_{42} is requested, and the example assumes that T_{42} has no dependencies with them, they can run in order, including in parallel, with respect to T_{42}.

This programming model does not give guarantees of relative order of execution with respect to the tasks that logically precede T that do not belong to the sets we have defined. Such tasks are all the descendants of the tasks in P_T that do not generate dependencies with T, that is, Descendants$(P_T - DP_T)$. The relation guaranteed with respect to these tasks is of mutual exclusion if there is at least a common element of conflict, i.e., one piece of data modifiable by either T or the considered task according to their respective lists of arguments. Going back to our example in Figure 4.1, the tasks in this situation with respect to T_{42} are T_{21} and T_{411}. In the execution shown in the figure they are requested before T_{42}, but in other runs any of these two tasks, of both, could be requested after T_{42}. This way, the relative order in which these three tasks will run cannot be predicted, but we are guaranteed that no pair of them with a conflict in their arguments will run in parallel.

The result of the described semantics is a simple rule of thumb for the programmer. Namely, a task T is guaranteed to fulfill dependencies with respect to the tasks requested before it, inside the same task, or before any of its ancestors, in the respec-

tive tasks where they were requested (condition(1.)), which is an informal definition for P_T. Task T also respects its dependencies with the tasks that are descended from those tasks in P_T that presented dependencies with T, DP_T (condition(2.)). For any other task U, it is guaranteed that U will be not be run in parallel with T if there is any memory position common to the arguments of T and U in which one of them writes. This behavior suffices in many situations. If it is not the case, the user has two possibilities to enforce the ordering between U and T. The most natural one is to express the dependency in an ancestor of U that belongs to P_T so that it becomes part of DP_T. The other one is to use one of the explicit synchronization mechanisms provided by our library, which are detailed in Section 4.1.3.

Listings 4.1 and 4.2 exemplifies how to enforce a proper ordering between tasks. In Listing 4.1 the master thread creates two tasks, `g` and `h` that carry no dependencies between them according to their arguments, so they can be run in parallel. During the execution of `g`, a new task `f` is requested that writes in a global object `a` used by task `h`. The system guarantees that `f` and `h` will not run in parallel, but it does not enforce any specific order of execution. Since `f` logically precedes `h` we probably want to make sure it runs before. This is achieved in a natural way by expressing

```
1  A a;
2
3  void f(A& a) {   }
4
5  void g() {
6    ...
7    spawn(f, a);
8    ...
9  }
10
11 void h(const A& a) { }
12
13 int main() {
14   spawn(g);
15   spawn(h, a);
16 }
```

Listing 4.1: Enforcing dependencies between tasks (a) Wrong

```
A a;

void f(A& a) {   }

void g(A& a) {
  ...
  spawn(f, a);
  ...
}

void h(const A& a) { }

int main() {
  spawn(g, a);
  spawn(h, a);
}
```

Listing 4.2: Enforcing dependencies between tasks (b) Right

the dependencies generated by `f` in the arguments of any ancestor spawned before `h`. Here such ancestor is `g`, giving place to the code in Listing 4.2.

4.1.2. Array support

The library behavior has been explained using generic data types, which can be standard types, user defined classes or arrays. The analysis performed by our library each time a parallel task is spawned treats all data types equally, checking the starting memory position and the size of each argument for overlaps with other variables. This permits expressing any kind of parallel computation, serializing tasks that access the same object when at least one of them writes to it. This raises an important question. Some objects are actually aggregates or containers of other objects, and the fact that multiple parallel tasks operate on them does not imply there are actually data dependencies among those tasks. For example, many parallel algorithms make use of arrays whose processing is distributed among different tasks that read and write to disjoint regions of these arrays. Therefore, just checking the full object or array is not a flexible strategy, as this would serialize

these actually independent tasks. One solution would be to distribute the data in the original array in smaller independent arrays so that each one of them is used by a different task, but this introduces non-negligible programming (and sometimes performance) overheads, and depending on the algorithm it is not always possible. Another solution, which is the one we have implemented, is to provide a data type that allows to express these arrays, to efficiently define subsections of them without copying data, and which is known to our dependencies analysis framework so it can retrieve the range of elements an array of this class refers to in order to check for overlaps.

In order to provide this support, we have developed a modified version of the `Array` class of the Blitz++ library [127]. Blitz++ implements efficient array classes and operations for numeric computations. Its `Array`, illustrated in Listing 4.3, provides multiple indexing schemes that allow to define sub- arrays that reference a bigger matrix (i.e. they point to the same data). Our `Array` class, derived from the one provided by Blitz++, enables our task spawn framework to check for overlapping subarrays that reference the same block of memory.

An example of a typical usage of the `Array` class is shown in Listing 4.4. This code subdivides the multiplication of two square matrices of `N`×`N` elements in `BLK`×`BLK` parallel tasks, each one being in charge of computing one portion of the output array `result`. The multiplication itself is performed in function `mxm`, which retrieves the dimensions of the arrays using their interface and accesses their scalar elements using operator `()`. The aim of this example is to illustrate the simple and powerful interface offered by `Array`s. A high-performance implementation should obtain the pointers from the arrays involved in the multiplication and invoke a specialized function such as `gemm` from BLAS. This is in fact the way we developed the applications used in the evaluation in Section 4.3.

```
1 // Two dimensional matrix of 64x64 floats
2 Array<float, 2> array(64, 64);
3
4 // Subarray from position (10,0) to position (20, 30)
5 Array<float, 2> subarray = array(Range(10, 20), Range(0, 30));
```

Listing 4.3: Example of definition of an array and a subarray.

```
void mxm(Array<float, 2>& result,
         const Array<float, 2>& a,
         const Array<float, 2>& b)
{
  const int nrows = result.rows();
  const int ncols = result.cols();
  const int kdim = a.cols();

  for(int i = 0; i < nrows; i++) {
    for(int j = 0; j < ncols; j++) {
      float f = 0.f;
      for(int k = 0; k < kdim; k++)
        f += a(i, k) * b(k, j);
      result(i, j) = f;
    }
  }
}

...
for(int i = 0; i < N; i += N / BLK) {
  int limi = (i + N / BLK) >= N ? N : (i + N / BLK);
  Range rows(i, limi - 1);
  for(int j = 0; j < N; j += N / BLK) {
    int limj = (j + N / BLK) >= N ? N : (j + N / BLK);
    Range cols(j, limj - 1);
    spawn(mxm, result(rows, cols), a(rows, Range::all()), b(Range::all(), cols));
  }
}
...
```

Listing 4.4: Usage of the `Array` class to enable the parallel processing of independent tasks.

4.1.3. Explicit synchronization facilities

DepSpawn also provides three functions to control the synchronization process at a lower level so that the programmer can make some optimizations:

- `void wait_for_all()` makes the current thread wait until all the spawned tasks finish. Its intended use is to serve as a barrier where the main program

can wait while the spawned processes do their operations.

- `void wait_for(Type vars...)` provides a more fine grained synchronization, where the current thread only waits for the tasks that generate dependencies on the variables specified as arguments to finish. There can be an arbitrary number of these variables and they can be of different types.

- `release(Type vars...)` can be used by a spawned task to indicate that the processing on some variables has ended, so that the dependent tasks can begin to run before this task actually finishes its execution.

4.1.4. Implementation details

Section 1.4 explained why we have chosen C++ and TBB for the implementation of our libraries. For DepSpawn, there are additional characteristics of this environment that were needed for the implementation:

- The nuclear idea of our library is to represent the parallel tasks by means of functions that express all their dependencies through their arguments. As a result, the function outputs should be provided through their arguments. This requires either resorting to pointers, which is the only option in C, or the ability to pass arguments by reference. Since it in unfeasible to automatically analyze dependencies among pointers using a library for the reasons explained in Section 4.1.1, we had to choose a language that provides pass by reference, which is the case of C++.

- We wanted our library to be as automated and to require as minimal user intervention as possible. This way, we preferred to use a language such as C++, with metaprogramming capabilities that allow a library to automatically analyze the types of the arguments of a function. In a language without this ability, the user would have to explicitly indicate to the library the number of arguments, as well as their intention and size, for each function to spawn.

- C++ templates allow to move many computations from runtime to compile time, particularly those related to types, which play an important role in this library, resulting in improved performance.

- An object oriented language that allows operator overloading is required to implement a class such as `Array` providing a nice notation. If the language further enables template classes, as C++ does, a single implementation of the class allows to provide support for generic arrays of any underlying type and different numbers of dimensions.

Another important feature of our library is that it does not give place to busy-wait situations. Rather, tasks with pending dependencies are stored so that they will be automatically launched to execution once all their dependencies are fulfilled. After storing such tasks, the corresponding threads of execution that encountered those requests proceed to the next instruction following the `spawn` invocation without further delays.

The data associated to the tasks and their dependencies are stored in structures that are internal to the library and which are updated each time a spawn is made or a tasks finishes. These structures are shared by all the threads so that all of them can keep track of the current set of dependencies to abide by and to remove the ones generated by each task when its execution finishes. This way, there is not a control thread in charge of these structures and deciding the tasks to spawn in each moment. Rather, all the threads operate on these structures, of course with proper synchronization, and launch new tasks to execution when their dependencies are satisfied.

Regarding the threading and load-balancing mechanism, our library is built on top of the low level API of the Intel Threading Building Blocks (TBBs) [112] library. This library has already been discussed in Section 1.4.

Another important component that this library is built upon is the Boost libraries [17]. These libraries provide a wide spectrum of advanced C++ utilities, ranging from different types of smart pointers to full fledged parallel graph structures. The main component we used to implement our library is the metaprogramming module MPL, which provides useful templates for the compile and runtime analysis of types in C++ programs, allowing a level of reflection that the language by itself does not provide. Other componentes used were `function_types` and *tttype_traits*, which facilitate the analysis of the types of functions and their arguments.

4.2. Tested algorithms

We first tested the correctness of our implementation with synthetic toy programs that covered all the possible combinations of task dependencies based on their input and output arguments. Then we have implemented several algorithms to test its performance and programmability. The next subsections briefly explain these algorithms.

4.2.1. N-body simulation using Barnes-Hut

This force-calculation algorithm employs a hierarchical data structure, called an quadtree, to approximately compute the force that the n bodies in a system induce upon each other. The algorithm hierarchically partitions the plane around the bodies into successively smaller cells. Each cell forms an internal node of the quadtree and summarizes information about the bodies it contains. The leaves of the quadtree are the individual bodies. This hierarchy reduces the time to calculate the force on the bodies because, for cells that are sufficiently far away, it suffices to perform only one force calculation with the cell instead of performing one calculation with each body inside the cell.

The algorithm has three main phases that run in sequence for each time step, until the desired ending time is reached. These phases are: (1) creating the quadtree,

```
1  barnes-hut {
2    Bodies[N];
3    while final time not reached {
4      spawn(create_quad-tree, Bodies);
5      for each block from Bodies:
6        spawn(compute_forces, block)
7      spawn(update, Bodies)
8    }
9    wait_for_all();
10 }
```

Listing 4.5: Pseudocode of the parallel implemenation using `spawn` of the Barnes-Hut algorithm

(2) computing the forces acting on each body, (3) updating the state of the system. The main computation load is in phase 2, whose computations can be done in parallel. Although phase 3 can be parallelized too, this is usually not considered because it is very lightweight.

The parallelization of this algorithm with `spawn` is quite simple: it is only needed to distribute the bodies in a block fashion and spawn the computation methods, as shown in the pseudocode in Listing 4.5. It must be noted however that load balancing, such as the one provided by our library, may play an important role on the performance of the parallel implementations of this algorithm. The reason is that the traversal of the quadtree performed in stage (2) of the algorithm does not have the same cost for each body. Namely, the traversal is deeper and requires more computations for the bodies located in more densely populated regions. For this reason our implementations overcompose the parallel loop in more tasks than available cores and let the underlying framework perform load balacing between the available cores. Notice also how the other stages are automatically synchronized thanks to our library, i.e., they only run when they are free of conflicts with preceding tasks.

4.2.2. LU decomposition

LU decomposition factorizes a matrix as the product of a lower triangular matrix and an upper triangular matrix. It is a key part of several numerical algorithms as the resolution of linear equations or computing the determinant of a matrix. The LAPACK [8] library contains a blocked implementation of this algorithm, represented in Figure 4.2 with its pseudocode in Listing 4.6, in which `A(i,j)` refers to the block in row `i` and column `j` in which the input matrix `A` has been divided. The algorithm progresses through the main diagonal of the matrix, and in each step it performs four operations: 1.- the LU decomposition of the diagonal block is computed using the unblocked version of the algorithm (`dgetf2`) (Line 2), 2.- it swaps the rows of the matrix from the diagonal to the end, according to the pivots retuned by the previous step (Line 4), 3.- it computes the solution of $A \times X = B$, being A the block on the diagonal and B the rest of the row to the end of the matrix (Line 9), and 4.- it multiplies the row and column blocks to obtain the next square

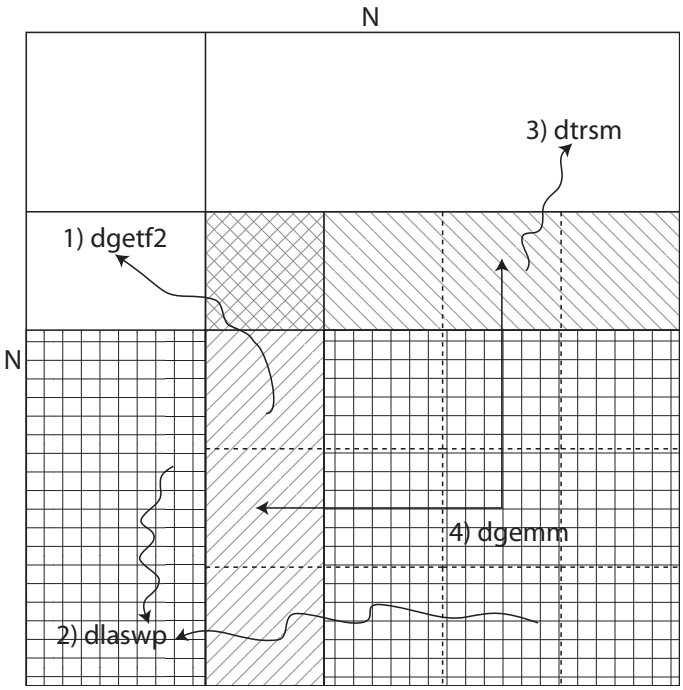

Figure 4.2: Blocked algorithm for computing the LU decomposition of a square matrix

submatrix (Line 13).

The parallelization strategy for this algorithm consist in subdividing the most expensive operations in blocks, in this case `dtrsm` but mainly `dgemm`. These blocks are represented by the dashed lines in Figure 4.2. Each block is assigned to a different task to perform the required operation. The `for_range` construction in Lines 9 and 13 represents a template function provided by our library whose purpose is to automatically divide an input range of blocks in smaller subblocks with the optimal size, so neither too many nor too few spawns are called.

4.2.3. Cholesky decomposition

Cholesky decomposition takes a matrix and computes its factorization as a lower triangular matrix and its conjugate transpose. The blocked version of the algorithm follows a structure similar to LU. First, the diagonal block is computed with the non-blocked version of the algorithm. Then, the remaining matrix is computed

```
1  for (j = 1; j < num_blocks; j ++) {
2    dgetf2(A(j:num_blocks, j), P);
3
4    update_pivots(P);
5    dlaswp(A(1:num_blocks, 1:j-1), P);
6    dlaswp(A(1:num_blocks,
7             j+1:num_blocks), P);
8
9    for_range(i in j+1:num_blocks) {
10     spawn(dtrsm, A(j, j), A(j, i));
11   }
12
13   for_range(i1 in j+1:num_blocks,
14             i2 in j+1:num_blocks) {
15     spawn(dgemm, A(i1, j),
16                  A(j, i2),
17                  A(i1, i2));
18   }
19   wait_for_all();
20 }
```

Listing 4.6: Pseudocode of the LU decomposition

multiplying the resulting blocks. Thus, the parallelization pattern is similar to the one used for LU, spawning tasks for the different sub-blocks of these operations.

The pseudocode of this algorithm is shown in Listing 4.7, in which again `A(i,j)` refers to a block. The algorithm has two basic steps: in Lines 2 and 3, the block of the diagonal is processed with an unblocked version of Cholesky, and from Line 5 to the end the remaining matrix is computed and prepared for the next iteration.

4.2.4. Sylvester equations resolution

The Sylvester equation [14], commonly found in control theory, is the equation of the form $AX + XB = C$ where A, B, C, X are $n \times n$ matrices, being X the unknown. We use $X = \Omega(A, B, C)$ to represent the solution to the equation. In particular, we focus on the triangular case, where both A and B are upper triangular matrices. The solution of the triangular case arises as an intermediate subproblem in the

```
for (j = 1; j < num_blocks; j ++) {
  dsyrk(A(1:num_blocks,j), A(j, j));
  dpotf2(A(j, j));

  for_range(i in j:num_blocks) {
    spawn(dgemm, transpose(A(1:j-1, j)), A(1:j-1, i), A(j, i));
  }

  for_range(i in j+1:num_blocks) {
    spawn(dtrsm, A(j, j), A(j, i));
  }
  wait_for_all();
}
```

Listing 4.7: Pseudocode of the Cholesky decomposition

Sylvester equation solver described in citeSYLVESTER. FLAME derives a family of blocked algorithms [60]. The result of X is stored in C. The algorithm is a hybrid of iterative and recursive algorithms. In this case, each of the blocks can be assigned to a different task, and the dependency detection system will take care of the order of execution to provide the correct result.

The computation of the solution for the Sylvester equation is done multiplying and recursively solving the blocks of the matrix. Listing 4.8 shows this process: the algorithm divides the matrices in 9 blocks, which change sizes as the algorithm progresses; then, for each block, the required operation is invoked, `syl` for recursively solving or `mul` to multiply two blocks. This algorithm has a complex dependency graph, shown in Figure 4.3, and it is the least scalable of the examples we tested.

4.3. Evaluation

In order to evaluate our approach we parallelized the algorithms described in the preceding Section using both our library and OpenMP [99], a standard high-level tool for the development of parallel applications in multicore systems. The performance tests were performed in a PC with an Intel i7 950 processor (with 4 cores and Hyperthreading) and 6GB of RAM, as well as in a server with 2 Intel

```
while(size(A22) > 0) {
  divide matrix;

  spawn(syl, A11, B00, C10);  // 1
  spawn(syl, A22, B11, C21);  // 2
  spawn(mul, C11, A12, C21);  // 3
  spawn(mul, C11, C10, B01);  // 4
  spawn(syl, A11, B11, C11);  // 5
  spawn(mul, C00, A01, C10);  // 6
  spawn(mul, C01, A01, C11);  // 7
  spawn(mul, C01, A02, C21);  // 8
  spawn(mul, C22, C21, B12);  // 9
  spawn(mul, C12, C10, B02);  // 10
  spawn(mul, C12, C11, B12);  // 11
}
```

Listing 4.8: Pseudocode of the Sylvester equations solver

Xeon E5620 quad-core processors and 16 GB of RAM. The compiler used was g++ v. 4.6.3 using -O3 optimization level.

For the N-body simulation we used a system of 100 000 bodies and simulated 1000 iterations. The speedups obtained in these experiments are shown in Figs. 4.4 and 4.5. Figures 4.6 to 4.11 show the performance of the considered linear algebra algorithms, using different combinations of hardware, libraries, and parallelization methods. Our library is compared with OpenMP and a purely sequential optimized version of the algorithms, using both the standard BLAS implementation [94] and the GotoBLAS2 library [58]. The rank of the double-precision matrices used in these tests is 8192. The i7 results for 8 cores actually correspond to the usage 4 cores with 2 threads per core thanks to the hyper-threading. It is well-known that the second thread per core provided by hyper-threading typically only provides between 5% and 20% of the performance of a real core. In fact this additional thread decreases performance for many applications due to the conflicts between the threads working sets in the core caches when large data sets are manipulated. This is the reason for the reduced performance for LU and Cholesky using 8 cores in the i7. As a matter of fact, the lack of optimizations in the usage of the memory hierarchy is a critical reason behind the poor performance, as well as the lack of scalability in the

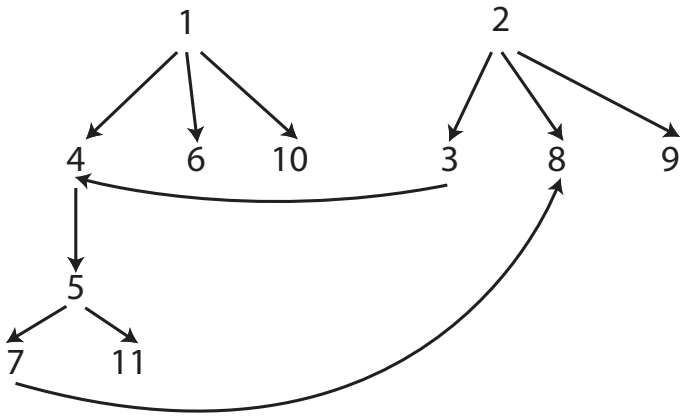

Figure 4.3: Dependencies of the Sylvester equation solver. Nodes refer to calls in Listing 4.8

i7, of the implementations based on the standard BLAS distribution. In the case of Sylvester, given the large number of stages of the algorithm that have no parallelism and the limited maximum number of parallel tasks available (see Figure 4.3), the small scalability was to be expected for any implementation.

The `spawn`-based version usually matches or outperforms the OpenMP version. We find two reasons for this. First, our library allows a task to run exactly as soon as its dependences have been satisfied. In OpenMP it is impossible to specify with such a fine grain the dependencies between tasks in different loops. Rather, synchronizations such as global barriers are required, which results in threads being idle more often. The other reason is that the task creation, scheduling and load balancing mechanisms provided by the OpenMP implementation can be less optimized and sophisticated than the ones that TBBs provide to our library [98][43].

We have also performed a comparison in terms of programmability of the codes parallelized with our library and OpenMP using the SLOCs, programming effort and cyclomatic number metrics described in Section 2.3.1. The results are shown in Table 4.2. Usually, our library achieves better results than OpenMP for any pro-

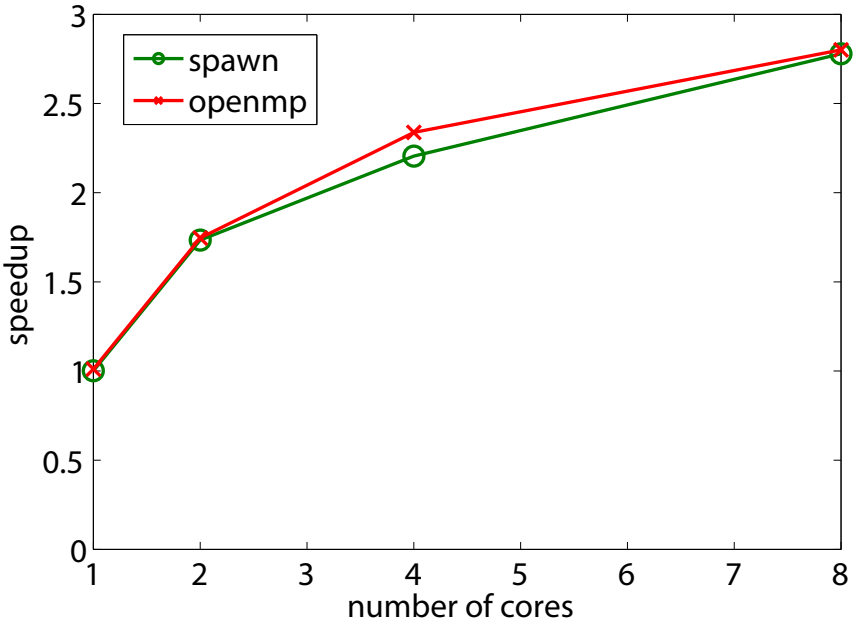

Figure 4.4: N-body simulation with Barnes-Hut algorithm in the i7 system

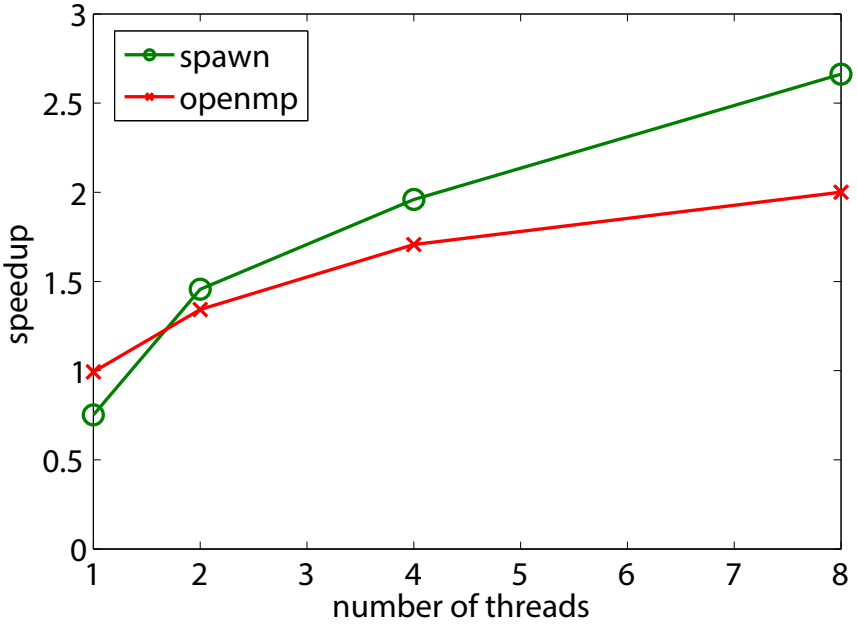

Figure 4.5: N-body simulation with Barnes-Hut algorithm in the Xeon system

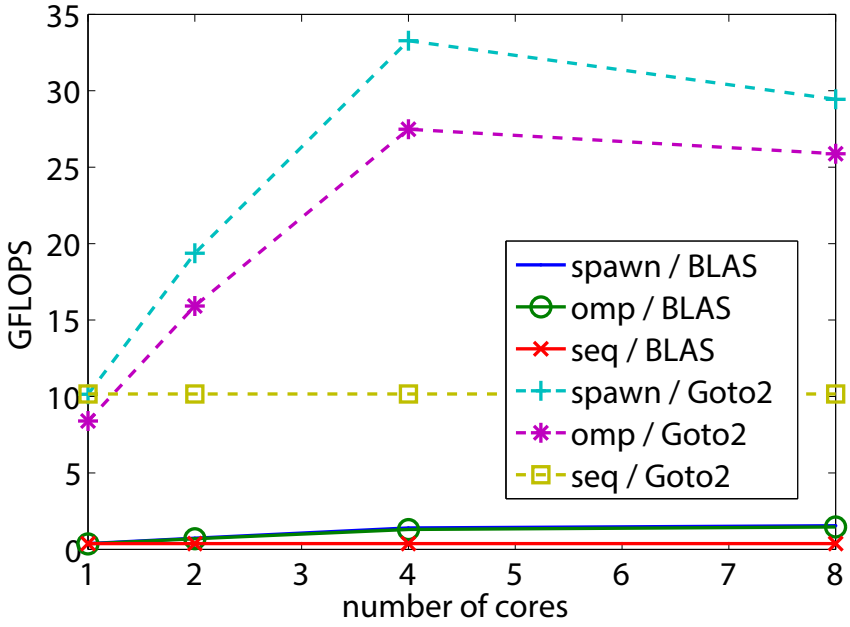

Figure 4.6: LU decomposition in the i7 system

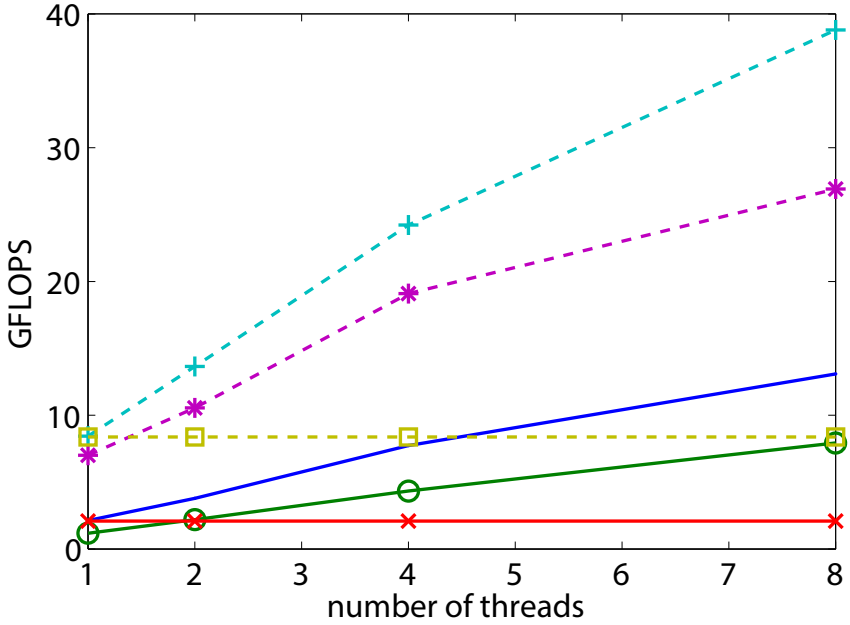

Figure 4.7: LU decomposition in the Xeon system

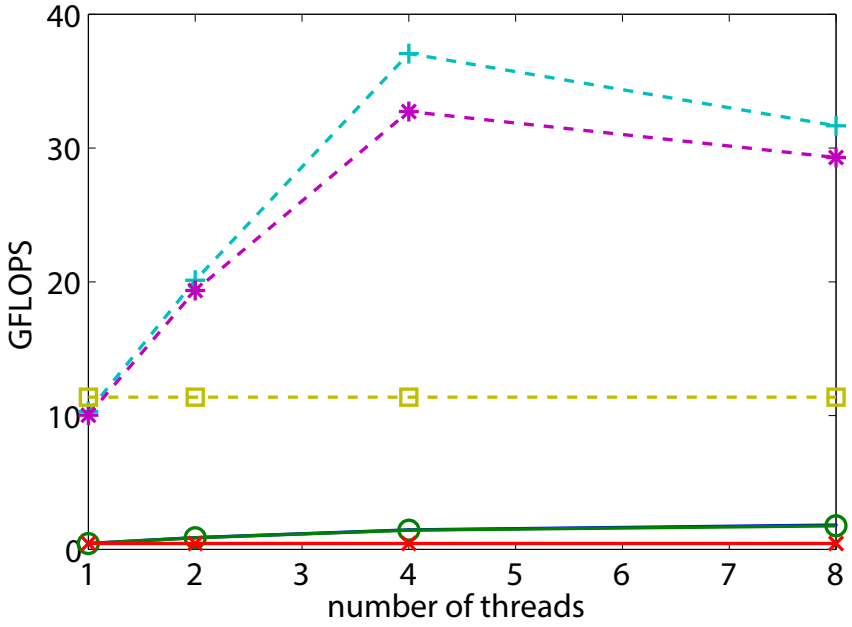

Figure 4.8: Cholesky decomposition in the i7 system

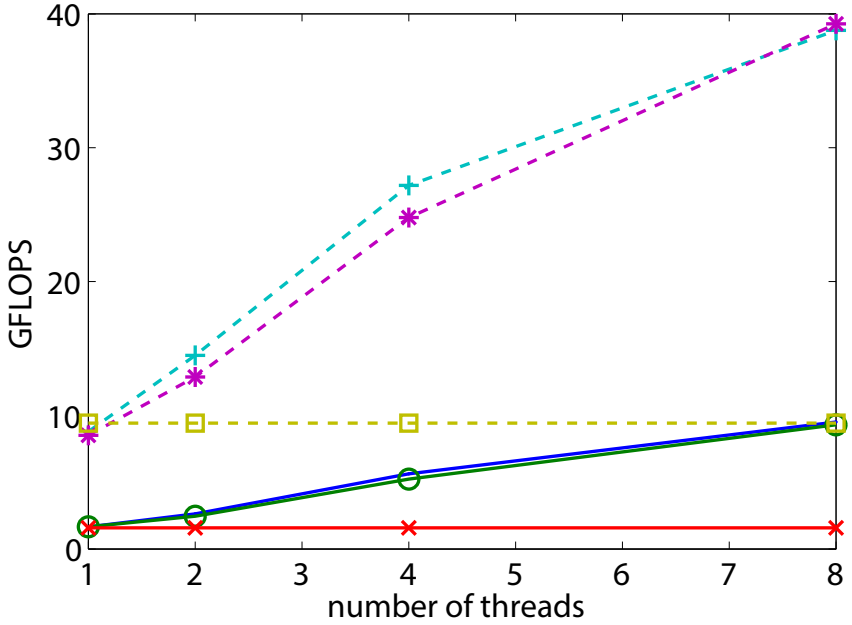

Figure 4.9: Cholesky decomposition in the Xeon system

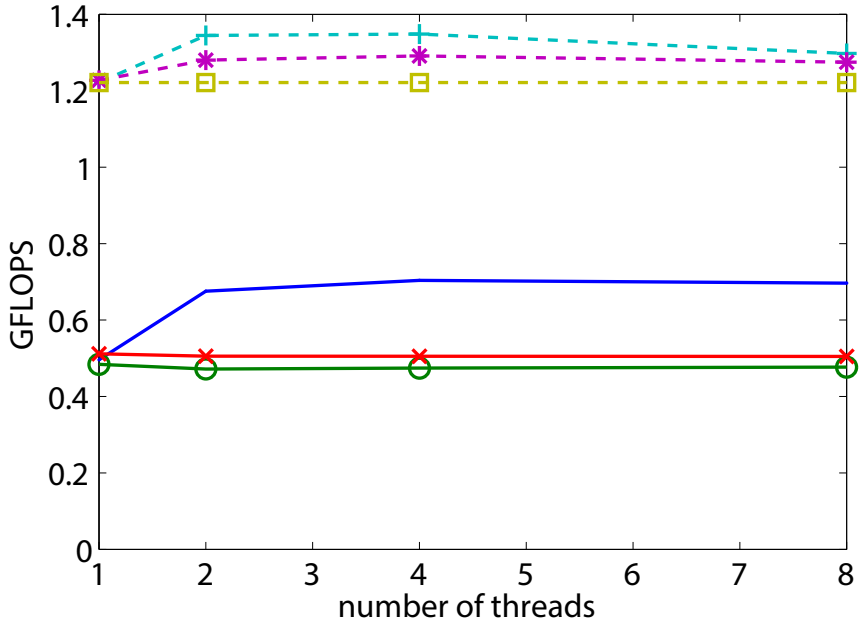

Figure 4.10: Sylvester equations in the i7 system

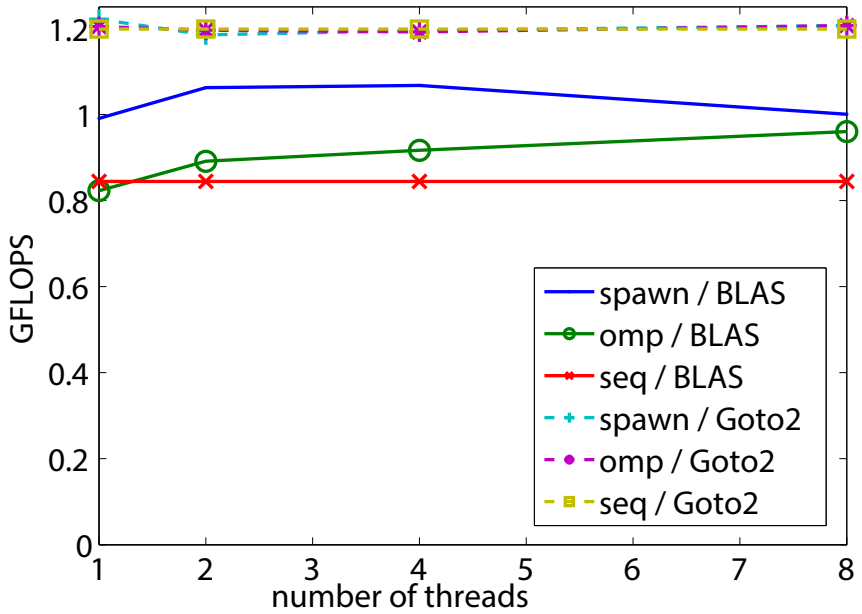

Figure 4.11: Sylvester equations in the Xeon system

Table 4.2: Programmability metrics

Application	version	SLOCs	Programming Effort	Cyclomatic #
Barnes-hut	spawn	214	919 352	36
	OpenMP	210	817 543	36
LU	spawn	94	349 770	12
	OpenMP	95	685 945	17
Cholesky	spawn	115	620 087	12
	OpenMP	116	796 005	16
Sylvester	spawn	98	789 667	3
	OpenMP	101	849 096	6

grammability metric. An important reason is that our `Array` class greatly simplifies the programming of array- based algorithms compared with the manual management of pointers and dimensions required by the standard approach. And this is the case even when we have counted the functions that unpack the pointers and matrices dimensions from the `Array`s to make the proper calls to the underlying BLAS implementations as part of the programming effort in Table 4.2. Since these are very typical matrix operations, they could well be included as part of our library, therefore further strongly improving all these metrics. The programmability also benefits from the notation required by our library, which is way terser than OpenMP. With `spawn`, one simply adds this word in the line of a procedure invocation that should be run as a parallel task, while OpenMP often requires separate multiword directives for the creation of parallel regions and individual tasks. In Barnes-Hut, however, our approach yields somewhat worse programmability statistics. The reason is that in this application the computational functions subject to parallel invocations were not generic functions, but class methods, and our current implementation of `spawn` does not support them in a straightforward way. This is simply solved adding functions that receive the object and the method arguments and which perform the method invocation, so that `spawn` is applied to these auxiliary functions.

4.4. Related work

The need for explicit synchronizations is proportional to the flexibility in the patterns of computation and parallelism supported by a programming paradigm or

tool. Functional programming, by avoiding state and mutable data, can provide referential transparency, so that the result of a function evaluation only depends on its arguments. This property allows in principle to evaluate in parallel any subexpression, as any order of evaluation yields the same result. Nevertheless, the exhaustive exploitation of all this implicit parallelism would involve much overhead. For this reason, in practice functional languages provide mechanisms to explictly label those computations whose parallel evaluation can be advantageous [124]. While some approaches [10][62][87] lead to explicit communications and synchronizations, in others the user just identifies the parallel tasks, letting the compiler or runtime take care of the low level details.

Data-parallelism, which applies a single stream of instructions in parallel to the elements of one or several data structures, is the basis of some of the implicitly synchronized functional proposals [102][107]. Unfortunately, this strategy is too restrictive for many applications either semantically or in terms of performance. A greater degree of flexibility is provided by parallel skeletal operations [33][109][125][84][53][118], which specify the dependencies, synchronizations and communications between parallel tasks that follow a pattern observed in many algorithms. Their applicability is restricted thus to computations that fit the predefined patterns they represent.

Interestingly, there are also proposals [125][88][86][84] that, like DepSpawn, allow to express tasks with arbitrary patterns of dependencies while avoiding explicit synchronizations. The fact that they are implemented in purely functional languages with lazy evaluation makes their programming strategy and the difficulties faced by the programmer very different from those of DepSpawn, which is integrated in an imperative language with the usual semantics. This way, their users have to deal with laziness, which hinders effective parallelization, and they have often to enforce sequential ordering, which are problems inexistent in our environment. Also, their functions cannot modify their inputs; rather they just return a result, whose usage expresses the dependency on the function, and which cannot be overwritten by other functions due to the immutability of data. This implies that once a function finishes, all the subsequent tasks that use its result can run in parallel, as they can only read it, there being no need to track potential ulterior modifications of its value. On the contrary, DepSpawn functions express their dependencies only

through their arguments, which can be inputs, outputs or both, while the return value is either inexistent or disregarded. Also, since the function arguments can be both read and/or written by arbitrary functions, the result(s) of a task can be overwritten by other tasks, leading to more complex patterns of dependencies on each data item than in a purely functional language. Finally, those approaches do not provide explicit synchronization facilities or classes similar to `Array` within their implicitly synchronized frameworks.

Dataflow programming [40][41] is another paradigm with links to our proposal. The reason is that it models programs as directed graphs whose nodes perform the computations and whose edges carry the inputs and outputs of those computations, thus interconnecting data-dependent computational nodes. This view promotes the parallel execution of independent computations, which are only run when their data dependencies are fulfilled, very much like DepSpawn tasks. Under this paradigm a DepSpawn task can be viewed as a node with input edges for its input arguments, and output edges for its output arguments. The node would only be triggered when there were data in all its input edges, and it would generate a result in each one of its output edges. Arguments that can be both inputs and outputs would be represented with edges both entering and leaving the node. If several tasks had a dependency on a given output of a task, the node of this task would be connected to each one of the dependent nodes with a separate output edge labeled with the output argument name. If the communication between DepSpawn tasks took place by means of individual copies, mimicking the behavior of edges that carry the data, this is all that would be needed. However, communication actually takes place through shared modifiable variables. This implies that if we want to represent a DepSpawn program with a dataflow graph whose tasks have the same set of legal schedules, we must prevent tasks that can write to a variable from beginning their execution while any preceding reader has not finished, even when there is no actual flow of data between them. This is achieved by connecting tasks that only read a given argument with output edges associated to it that link them with the next task that can modify that argument. This way the writer cannot be triggered until the reader finishes and puts a value in the connecting edge, even if the value read from the edge is discarded. Figure 4.12 illustrates all the situations described above with a small piece of code and its corresponding dataflow graph. The edge `a′` has been marked with an apostrophe to indicate that there is no actual flow of data, but a

signal to prevent `f(c, a)` from modifying `a` while `f(a, b)` is still working on it. The figure also illustrates why a node must be used per invocation/task rather than by function, as, for example, otherwise it would be impossible to run parallel instances of a function.

While all of them have in common that data in the input edges of a node trigger its computation, which in turn generates new data in its output edges, there are several models of dataflow networks with different assumptions and semantics [81]. A DepSpawn application in which each task is deterministic and tasks only communicate through their arguments can be modeled as a Kahn Process Network (KPN) [71], which is a network composed by deterministic sequential processes that communicate through one-way unbounded FIFO channels. In fact since data are always consumed and produced by means of a single argument, which is of an aggregate type such as `Array` when several data items are involved, a network of tasks generated by DepSpawn can be more accurately modeled as a Synchronous Dataflow (SDF) [80], a restriction of KPN in which nodes consume and produce a fixed number of data items per firing in each one of their edges. Furthermore, Depspawn dataflow graphs are homogeneous SDFs, as all nodes produce or consume a single sample on each input or output arc when invoked. The properties of the associated computation graphs have been analyzed in [73][113][35].

Notice that the aforementioned characterization is also valid when tasks spawn children tasks, provided that these children are also deterministic and only communicate with other tasks through their arguments. This is thanks to the fact that

```
void g(A& a, A& b, A& c, A& d) {
   ...
}

void f(const A& x, A& y) {
   ...
}

spawn(g, a, b, c, d);
spawn(f, a, b);
spawn(f, c, d);
spawn(f, b, d);
spawn(f, c, a);
wait_for_all();
```

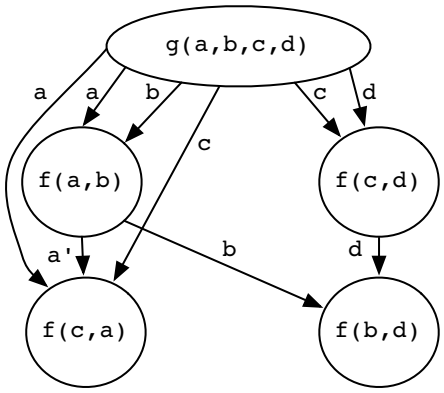

Figure 4.12: Small DepSpawn code and associated dataflow graph.

DepSpawn tasks wait for all the preceding tasks with which they have dependencies and their descendants.

Programs developed under the usual imperative languages can also avoid explicit synchronizations by resorting to data-parallel approaches [119][70][48][39] and parallel skeletons [34][46][112][52][53], at the cost again of restricting the patterns of parallelization. Because of their limitations, users are often forced to resort to lower level approaches [20][110][97][22][112][82][21][99] seeking more flexibility. The downside is that these tools hurt productivity, being explicit synchronizations one of the reasons. Some of them [110][112][82][21][99] have in common with DepSpawn that they allow to build tasks expressed as functions, although often with a more convoluted syntax, and that they let the runtime manage the execution of these tasks using advanced techniques such as work-stealing. In fact, as explained in Section 4.1.4, DepSpawn in built on top of TBBs [112]. However, none of them can automatically track and enforce the dependencies among the tasks. This way, they all require explicitly synchronizing the tasks by linking each task with its successors or predecessors, establishing synchronization points, or accessing futures attached to variables that represent the tasks. Achieving a behavior similar to DepSpawn, in terms of performance, minimal interface and automatic fulfillment of data dependencies by means of these tools is a non trivial task.

The idea of exploring arbitrary out-of-order execution of tasks by relying on the dependences among them has already been explored in the context of libraries and compilers. This way, Supermatrix [25][27][26] provides a library exclusively focused on linear algebra algorithms that is able to execute its parallel tasks out-of-order respecting the serial semantics by means of a task dependency analysis. Since, contrary to our library, its aim is not general, it does not define any particular programming model either.

As for compiler-based approaches, the Star Superscalar (StarSs) programming model, with implementations for different devices such as the Cell broadband engine [105] or general SMP computers [101], similarly to our library, seeks to provide general out-of-order execution of tasks based on data dependencies. Nevertheless, this paradigm, which has led to the proposal of extensions to OpenMP [44][13], requires the user to explicitly annotate such dependencies by means of compiler directives in the code. This involves not only analyzing the code to establish which

are the inputs and outputs of each task, but also which is the exact size of the area of the arrays pointed by the pointers used in these tasks, as well as the extension of the region accessed by each pointer. Our library provides elegant solutions to these problems by directly extracting the information from the function parameters or from the `Array` objects provided, resulting in clearer and less error-prone codes.

There are also important differences in the programming model of both approaches. Namely, in the programming model supported by these compiler directives, dependencies are only detected inside the scope of the same parent task. This way if there is any piece of data on which there can be carried dependencies that need to be considered by a task, it must be explicitly annotated in its parent. Obviously, this also implies in turn annotating all the ascendants up to the level where the potential dependency is generated. And these annotations also imply that those ascendants will have to abide by those dependencies, even when they are actually only needed for the bottom-level task we were initially considering. Nevertheless, under the programming model provided by our library, tasks automatically fulfill any dependencies generated not only in their parent task, but also in all of their ancestors, there being no need to apply those dependencies to any of those ancestors if they do not need them. This significatively increases the amount of parallelism that can be exploited in many situations. Other distinctive features of our library that improve this aspect are the possibility of releasing (some) dependencies before a task finishes, or blocking a task at some arbitrary point in order to wait for a specific set of variables.

4.5. Conclusions

In this chapter we have presented DepSpawn, a new library for parallel programming that provides very flexible patterns of parallelism without the need of explicit synchronizations. Using advanced features of C++11, our library is able to analyze the parameters of arbitrary functions and detect dependencies between them. With this information it schedules their parallel execution while respecting their dependencies. We have also provided a clear description of the programming model enabled by our library as well as a comparison with a standard high-level approach to parallelize applications in multicore systems like OpenMP. The results

obtained are very satisfactory, both in terms of performance and programmability.

Chapter 5

Conclusions

For decades the scientific community primarily focused on the parallelization of codes with regular control flows, structures and access patterns, as they are easily found in the scientific and engineering applications to which parallelism was mostly restricted until multicores made it ubiquitous, thus sparking the interest in the parallelization of every kind of application. As a result, while regular perfectly data-parallel applications are well understood and supported, algorithms that are better described in terms of more complex patterns of parallelism often require programmers to resort either to manual parallelization using low-level tools, which is error-prone and costly, or to transactional solutions that require specific hardware or present potentially large overheads. This thesis is an attempt to better understand some of these problems and to provide tools that improve their programmability while providing reasonable performance.

In this dissertation we have considered three kinds of problems whose parallelization does not adjust well to the most commonly used tools for different reasons: the divide-and-conquer pattern, the amorphous data-parallel algorithms, and the applications based on tasks with arbitrary patterns of dependences. As a result of our analysis we have provided a library-based solution well suited for each one of them in shared-memory systems. Our libraries are developed in C++, as it is a very popular language that provides both high performance and excellent tools to express high-level abstractions. The underlying framework used by our proposals to create and manage the parallelism is the Intel Threading Building Blocks (TBB) library [112],

as it is widely available and it showed better behavior than other alternatives in the tests we performed before developing the final version of our libraries. Thanks to this and to the multiple optimizations applied in our proposals, the performance that they achieve is competitive with that of existing approaches, while programmability improvements are observed in the vast majority of the experiments.

Out of the three problems tackled, the first one, which is the parallelization of the traditional divide-and-conquer pattern, is the most well-known one. Despite this fact, and the enormous relevance of this pattern, we did not find a flexible skeleton based on high-level abstractions for its implementation in shared-memory systems. The need for such skeleton has been motivated in this thesis based on an analysis of the problems of its implementation by means of the most similar skeleton provided by the most widely used skeleton library nowadays, which is the Intel TBB. Our proposal, which we have called `parallel_recursion`, uses one object to provide information on the structure and decomposition of the input problem and another one to provide the operations to perform. Using our skeleton resulted in codes between 2.9 and 4.6 times shorter in terms of SLOCS than the TBB implementations when only the portion of the codes affected by the parallelization was considered. Even when considering the whole application and the more accurate programming effort metric, which takes into account the number and variety of symbols used in the code, `parallel_recursion` required 14.6% less effort than the standard TBB for the parallelization of these algorithms. We have also noted that in the specific case of algorithms that do not need any function to combine the results of their subproblems to build the final result and that are based on arrays, which naturally fit the TBB ranges in which TBB templates are based, TBB provided better programmability metrics than our library, requiring 11.7% less effort. As for performance, our skeleton performed on average better than both the TBB and OpenMP implementations in the two machines tested when automatic work partitioning was used, although manual granularity selection could allow TBB to outperform `parallel_recursion` in one of the machines.

Amorphous data-parallelism is the second and probably the most complex problem considered in this dissertation, given the highly irregular nature and the dynamically changing conditions that characterize the applications that fit this paradigm. This thesis proposes the parallelization of these problems by extending data-parallelism

with powerful abstractions [18] and applying the well-known concept of skeleton [57] to this new field. This way, our proposal is a skeleton called `parallel_domain_proc` that is based on the abstraction of a domain on which the elements to process are defined. Our skeleton uses this domain both to partition work, by recursively subdividing the input domain, and to detect potential conflicts among parallel computations, by testing the ownership of the elements to access by the current subdomain considered. The skeleton is totally agnostic with respect to the object that represents the irregular structure to process, just requiring that it can support concurrent updates from parallel tasks, and it has a few requirements on the API and semantics of the domain objects. In addition, the fact that the skeleton tests to detect conflicts are based on conditions computed on the elements to process, namely on their belonging to a domain, rather than on the usual lock-based strategies, avoids the busy waiting and contention problems usually associated to locks. Another advantage of our approach is that work-items are examined at most once per level of subdivision of the input domain, which provides a clear bound on the maximum number of attempts to process them. In our experiments the parallel versions developed using our skeleton required a maximum of 3% more lines of code than their sequential counterparts, while they used in fact even fewer conditional statements in the user code, which is reflected in a smaller cyclomatic number, thanks to the embedding in our library of several of the required tests and loops. As for performance, we have showed that in these applications it largely depends on many factors that our skeleton allows to adjust, such as the work decomposition policy, the granularity of the tasks or the data structures used. Finally, a qualitative comparison with the related work indicates that the speedups achieved with our library are on par with those achieved using other approaches, some of them being manual parallel implementations.

The third problem we have considered is the ability to express in the most convenient way tasks that must abide by arbitrary data dependencies so that such dependencies are automatically enforced. Our solution, called DepSpawn, requires to write such tasks as functions, which can be regular C++ functions, but also the new convenient C++11 lambda functions or `std::functions`, so that their inputs and outputs are solely provided by their list of parameters. These functions must be launched to execution using the provided function `spawn`, followed by their list of arguments. This effectively discovers which are the inputs and outputs of the

function and takes the necessary steps to ensure the function is only run when all its dependences are satisfied. The concrete semantics implemented by our library have been carefully described, and a special data type to support the parallel processing of different portions of arrays has been provided together with a few explicit synchronization facilities. Our evaluation reveals that the applications based on DepSpawn typically match or outperform the codes developed using OpenMP because it can run tasks just as soon as their individual dependencies are met and thanks to the advantages of TBB with respect to OpenMP. Just as in the other problems considered, our solution usually resulted in better programmability metrics than the codes parallelized with OpenMP. In addition, a detailed discussion that has examined both existing functional and imperative approaches has shown that DepSpawn is either more general or presents several programmability and performance advantages with respect to the previous proposals. In the case of the more related imperative approaches the reason is that they are either oriented to specific fields of application or they require more information from the users and present more restrictions in their applicability.

5.1. Future Work

As future work, the set of libraries can be expanded to address a wider set of parallel programing models. One interesting idea is to expand the backend of the libraries so they support distributed memory systems. Also, the configurability of the libraries can be increased, adding optional interfaces that allow the programmer to use her knowledge of the concrete problem to introduce hints that could improve load balancing and performance. This would be specially useful for the case of `parallel_domain_proc`, which uses complex data structures whose best processing strategy can not be fully known at compile time.

While the domain partitioning strategies provided for `parallel_domain_proc` in our library are reasonable generic approaches, they may be enhanced with the use of well-known graph-partitioners [74][100] and domain specific strategies. Also, methods to backup data to be modified so that they can be restored later automatically by the library if the computation fails can be added in order to support non cautious operations.

As for DepSpawn, a current limitation is that it cannot manage the return values of the functions used as tasks, which forces them to return all their results by means of their arguments. We plan to extend the support for functions that return values using the concept of futures. Namely, for these functions `spawn` would return a special object to hold the value returned by the function. Reading this object would conform an implicit synchronization point.

Bibliography

[1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable Graph Exploration on Multicore Processors. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] S. Aldea. *Compile-Time Support for Thread-Level Speculation*. PhD dissertation, Departamento de Informática, Universidad de Valladolid, 2014.

[4] S. Aldea, A. Estebanez, D. R. Llanos, and A. González-Escribano. A New GCC Plugin-Based Compiler Pass to Add Support for Thread-Level Speculation into OpenMP. In *Proc. 20th Intl. Euro-Par Conf. on Parallel Processing, Euro-Par 2014*, pages 234–245, 2014.

[5] S. Aldea, D. R. Llanos, and A. González-Escribano. Support for Thread-Level Speculation into OpenMP. In *Proc. 8th Intl. Conf. on OpenMP in a Heterogeneous World (IWOMP 2012)*, pages 275–278, 2012.

[6] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in Java. *Future Gener. Comput. Syst.*, 19(5):611–626, 2003.

[7] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Thomas, N. Amato, and L. Rauchwerger. STAPL: an adaptive, generic parallel C++ library.

In *Proceedings of the 14th international conference on Languages and compilers for parallel computing*, LCPC'01, pages 193–208, Berlin, Heidelberg, 2003. Springer-Verlag.

[8] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: a portable linear algebra library for high-performance computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, Supercomputing '90, pages 2–11, Los Alamitos, CA, USA, 1990. IEEE Computer Society Press.

[9] D. Andrade, B. B. Fraguela, J. C. Brodman, and D. A. Padua. Task-Parallel versus Data-Parallel Library-Based Programming in Multicore Systems. In *17th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2009)*, pages 101–110. IEEE Computer Society, 2009.

[10] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[11] R. C. Armstrong and A. Cheung. POET (Parallel Object-oriented Environment and Toolkit) and Frameworks for Scientific Distributed Computing. In *Proc. of 30th Hawaii International Conference on System Sciences (HICSS)*, pages 54–63, 1997.

[12] V. Çatalyürek, J. Feo, A. H. Gebremedhin, M. Halappanavar, and A. Pothen. Graph Coloring Algorithms for Multi-core and Massively Multithreaded Architectures. *Parallel Comput.*, 38(10-11):576–594, Oct. 2012.

[13] E. Ayguadé, R. M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. González, F. D. Igual, D. Jiménez-González, and J. Labarta. Extending OpenMP to Survive the Heterogeneous Multi-Core Era. *Intl. J. Parallel Program.*, 38(5-6):440–459, 2010.

[14] R. H. Bartels and G. W. Stewart. Solution of the matrix equation AX + XB = C [F4]. *Commun. ACM*, 15(9):820–826, sep 1972.

[15] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintana-Ortí, and R. A. van de Geijn. The Science of Deriving Dense Linear Algebra Algorithms. *ACM Trans. Math. Softw.*, 31(1):1–26, Mar. 2005.

[16] G. Bikshandi, J. Guo, C. von Praun, G. Tanase, B. B. Fraguela, M. J. Garzarán, D. Padua, and L. Rauchwerger. Design and Use of htalib – A Library for Hierarchically Tiled Arrays. In *Languages and Compilers for Parallel Computing*, volume 4382 of *Lecture Notes in Computer Science*, pages 17–32. Springer Berlin Heidelberg, 2007.

[17] Boost.org. Boost C++ libraries. http://boost.org.

[18] J. C. Brodman, B. B. Fraguela, M. J. Garzarán, and D. Padua. New abstractions for data parallel programming. In *First USENIX Conf. on Hot Topics in Parallelism (HotPar'09)*, pages 16–16, 2009.

[19] D. Buono, M. Danelutto, and S. Lametti. Map, reduce and mapreduce, the skeleton way. *Procedia CS*, 1(1):2095–2103, 2010.

[20] D. R. Butenhof. *Programming with POSIX Threads*. Addison Wesley, 1997.

[21] C. Campbell, R. Johnson, A. Miller, and S. Toub. *Parallel Programming with Microsoft .NET - Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, 2010.

[22] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. Introduction to UPC and Language Specification. Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, 1999.

[23] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *Queue*, 6(5):46–58, 2008.

[24] B. Chamberlain, S.-E. Choi, E. Lewis, L. Snyder, W. Weathersby, and C. Lin. The case for high-level parallel programming in ZPL. *Computational Science Engineering, IEEE*, 5(3):76 –86, jul-sep 1998.

[25] E. Chan, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. Supermatrix out-of-order scheduling of matrix operations for SMP and multi-core

architectures. In *Proc. 19th ACM symp. on Parallel algorithms and architectures*, SPAA'07, pages 116–125, 2007.

[26] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In *Proc. 13th ACM SIGPLAN Symp. on Principles and practice of parallel programming*, PPoPP'08, pages 123–132, 2008.

[27] E. Chan, F. G. Van Zee, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. Satisfying your dependencies with SuperMatrix. In *Proc. 2007 IEEE Intl. Conf. on Cluster Computing*, CLUSTER'07, pages 91–99, 2007.

[28] P. Charles, C. Donawa, K. Ebcioglu, C. Grothoff, A. Kielstra, C. von Praun, V. Saraswat, and V. Sarkar. X10: An Object-oriented Approach to Non-uniform Cluster Computing. In *Procs. of the Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) – Onward Track*, 2005.

[29] L. P. Chew. Guaranteed-quality mesh generation for curved surfaces. In *Proceedings of the ninth annual symposium on Computational geometry*, SCG '93, pages 274–280. ACM, 1993.

[30] P. Ciechanowicz, M. Poldner, and H. Kuchen. The Münster Skeleton Library Muesli - A Comprehensive Overview. Technical Report Working Papers, ERCIS No. 7, University of Münster, 2009.

[31] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 13–24, 2003.

[32] M. Cintra, J. F. Martínez, and J. Torrellas. Architectural support for scalable speculative parallelization in shared-memory multiprocessors. In *Proc. of the 27th intl. symp. on Computer architecture (ISCA)*, pages 256–264, 2000.

[33] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, 1991.

[34] M. Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30:389–406, March 2004.

[35] F. Commoner, A. W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *J. Comput. Syst. Sci.*, 5(5):511–523, oct 1971.

[36] D. E. Culler, A. Gupta, and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1997.

[37] M. Danelutto. Efficient Support for Skeletons on Workstation Clusters. *Parallel Processing Letters*, 11(1):41–56, 2001.

[38] M. Danelutto and M. Torquati. Loop Parallelism: A New Skeleton Perspective on Data Parallel Patterns. In *Parallel, Distributed and Network-Based Processing (PDP), 2014 22nd Euromicro International Conference on*, pages 52–59, Feb 2014.

[39] A. de Vega, D. Andrade, and B. B. Fraguela. An efficient parallel set container for multicore architectures. In *intl. conf. on Parallel Computing*, ParCo 2011, pages 369–376, 2011.

[40] J. B. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 362–376. Springer-Verlag, 1974.

[41] J. B. Dennis. Data Flow Supercomputers. *Computer*, 13(11):48–56, nov 1980.

[42] A. Dios, R. Asenjo, A. Navarro, F. Corbera, and E. Zapata. High-level template for the task-based parallel wavefront pattern. In *High Performance Computing (HiPC), 2011 18th International Conference on*, pages 1–10, Dec 2011.

[43] A. J. Dios, R. Asenjo, A. G. Navarro, F. Corbera, and E. L. Zapata. Evaluation of the task programming model in the parallelization of wavefront problems. In *12th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC 2010)*, pages 257–264, 2010.

[44] A. Duran, R. Ferrer, E. Ayguadé, R. M. Badia, and J. Labarta. A proposal to extend the OpenMP tasking model with dependent tasks. *Intl. J. Parallel Program.*, 37(3):292–305, June 2009.

[45] J. Enmyren and C. Kessler. SkePU: a multi-backend skeleton programming library for multi-GPU systems. In *4th intl. workshop on High-level parallel programming and applications*, HLPP '10, pages 5–14, 2010.

[46] J. Falcou, J. Sérot, T. Chateau, and J.-T. Lapresté. Quaff: efficient C++ design for parallel skeletons. *Parallel Computing*, 32(7-8):604–615, 2006.

[47] B. B. Fraguela, G. Bikshandi, J. Guo, M. J. Garzarán, D. Padua, and C. von Praun. Optimization techniques for efficient HTA programs. *Parallel Computing*, 38(9):465 – 484, 2012.

[48] B. B. Fraguela, J. Guo, G. Bikshandi, M. J. Garzarán, G. Almási, J. Moreira, and D. Padua. The Hierarchically Tiled Arrays programming approach. In *Proc. 7th Workshop on languages, compilers, and run-time support for scalable systems*, LCR '04, pages 1–12, October 2004.

[49] B. B. Fraguela, J. Renau, P. Feautrier, D. Padua, and J. Torrellas. Programming the FlexRAM Parallel Intelligent Memory System. *SIGPLAN Not.*, 38(10):49–60, June 2003.

[50] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-Oblivious Algorithms. In *FOCS '99: Procs. 40th Annual Symp. on Foundations of Computer Science*, page 285, 1999.

[51] A. Geist. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and engineering computation. MIT Press, 1994.

[52] C. H. González and B. B. Fraguela. A Generic Algorithm Template for Divide-and-Conquer in Multicore Systems. In *12th IEEE Intl. Conf. on High Performance Computing and Communications (HPCC 2010)*, pages 79 –88, sept 2010.

[53] H. González-Vélez and M. Leyton. A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, Nov. 2010.

[54] C. H. González and B. B. Fraguela. A framework for argument-based task synchronization with automatic detection of dependencies. *Parallel Computing*, 39(9):475–489, 2013.

[55] C. H. González and B. B. Fraguela. An algorithm template for domain-based parallel irregular algorithms. *International Journal of Parallel Programming*, 42(6):948–967, 2014.

[56] C. H. González and B. B. Fraguela. Enhancing and evaluating the configuration capability of a skeleton for irregular computations. In *23rd Euromicro International Conference on Parallel, Distributed and Network-based Processing*, 2015. accepted for publication.

[57] S. Gorlatch and M. Cole. Parallel skeletons. In *Encyclopedia of Parallel Computing*, pages 1417–1422. 2011.

[58] K. Goto. GotoBLAS2. http://www.tacc.utexas.edu/tacc-projects/gotoblas2.

[59] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI (2nd ed.): portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA, 1999.

[60] J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Softw.*, 27(4):422–455, dec 2001.

[61] M. Gupta and R. Nim. Techniques for run-time parallelization of loops. *Supercomputing*, pages 1–12, 1998.

[62] P. Haller and M. Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, Feb. 2009.

[63] M. H. Halstead. *Elements of Software Science*. Elsevier, 1977.

[64] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proc. of the 8th Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 58–69, 1998.

[65] T. Harris and K. Fraser. Language support for lightweight transactions. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, OOPSLA '03, pages 388–402. ACM, 2003.

[66] M. A. Hassaan, M. Burtscher, and K. Pingali. Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms. *SIGPLAN Not.*, 46(8):3–12, Feb. 2011.

[67] K. A. Hawick, A. Leist, and D. P. Playne. Parallel graph component labelling with GPUs and CUDA. *Parallel Computing*, 36(12):655–678, 2010.

[68] J. L. Hennessy and D. A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

[69] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300. ACM, 1993.

[70] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Commun. ACM*, 35:66–80, August 1992.

[71] G. Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress*, pages 471–475, Aug 1974.

[72] R. M. Karp. Probabilistic analysis of partitioning algorithms for the traveling-salesman problem in the plane. *Math. of Operations Research*, 2(3):209–224, 1977.

[73] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM J. Appl. Math.*, 14(6):1390–1411, nov 1966.

[74] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1):359–392, 1998.

[75] H. Kuchen. A Skeleton Library. In *Proc. 8th Intl. Euro-Par Conf. on Parallel Processing*, pages 620–629, 2002.

[76] M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Casçaval. How much parallelism is there in irregular applications? *SIGPLAN Not.*, 44:3–14, February 2009.

[77] M. Kulkarni, M. Burtscher, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *in IEEE International Symposium on Performance Analysis of Systems and Software*, pages 65–76, 2009.

[78] M. Kulkarni, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L. P. Chew. Optimistic parallelism benefits from data partitioning. *SIGOPS Oper. Syst. Rev.*, 42(2):233–243, 2008.

[79] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *SIGPLAN Not.*, 42(6):211–222, June 2007.

[80] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, sept 1987.

[81] E. Lee and T. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773 –801, may 1995.

[82] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. *SIGPLAN Not.*, 44(10):227–242, Oct. 2009.

[83] M. Leyton and J. Piquer. Skandium: Multi-core Programming with Algorithmic Skeletons. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 289–296, Feb 2010.

[84] R. Loogen, Y. Ortega-mallén, and R. Peña marí. Parallel functional programming in Eden. *J. Funct. Program.*, 15(3):431–475, May 2005.

[85] R. Lublinerman, S. Chaudhuri, and P. Cerný. Parallel programming with object assemblies. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 61–80, 2009.

[86] S. Marlow, P. Maier, H.-W. Loidl, M. Aswad, and P. Trinder. Seq no more: better strategies for parallel Haskell. *SIGPLAN Not.*, 45(11):91–102, Sept. 2010.

[87] S. Marlow, R. Newton, and S. Peyton Jones. A monad for deterministic parallelism. *SIGPLAN Not.*, 46(12):71–82, Sept. 2011.

[88] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. *SIGPLAN Not.*, 44(9):65–78, Aug. 2009.

[89] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming. In *Infoscale*, 2006.

[90] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming.* Addison-Wesley Professional, 2004.

[91] McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[92] M. Méndez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassaan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-driven optimizations for amorphous data-parallel programs. In *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10, pages 3–14. ACM, 2010.

[93] J. Meng, S. Che, J. W. Sheaffer, J. Li, J. Huang, and K. Skadron. Hierarchical Domain Partitioning For Hierarchical Architectures. Technical Report CS-2008-08, Univ. of Virginia Dept. of Computer Science, June 2008.

[94] National Science Foundation and Department of Energy. BLAS. http://www.netlib.org/blas/, 2011.

[95] A. Navarro, R. Asenjo, F. Corbera, A. J. Dios, and E. L. Zapata. A case study of different task implementations for multioutput stages in non-trivial parallel pipeline applications . *Parallel Computing*, 40(8):374 – 393, 2014.

[96] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global arrays: a portable shared-memory programming model for distributed memory computers. In

Supercomputing '94: Proc. of the 1994 Conf. on Supercomputing, page 340. IEEE Computer Society Press, 1994.

[97] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[98] S. Olivier and J. F. Prins. Comparison of OpenMP 3.0 and Other Task Parallel Frameworks on Unbalanced Task Graphs. *Intl. J. Parallel Program.*, 38(5-6):341–360, 2010.

[99] OpenMP Architecture Review Board. *OpenMP Program Interface Version 3.1*, July 2011. http://www.openmp.org.

[100] F. Pellegrini and J. Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

[101] J. Perez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *2008 IEEE Intl. Conf. on Cluster Computing*, pages 142 –151, oct 2008.

[102] S. Peyton Jones. Harnessing the Multicores: Nested Data Parallelism in Haskell. In *Proc. 6th Asian Symp. on Programming Languages and Systems*, APLAS'08, pages 138–138, 2008.

[103] K. Pingali, M. Kulkarni, D. Nguyen, M. Burtscher, M. Mendez-lojo, D. Prountzos, X. Sui, and Z. Zhong. Amorphous Data-parallelism in Irregular Algorithms. Technical Report TR-09-05, The Univ. of Texas at Austin, Dpt. of Computer Sciences, Feb. 2009.

[104] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The Tao of Parallelism in Algorithms. *SIGPLAN Not.*, 46(6):12–25, June 2011.

[105] J. Planas, R. M. Badia, E. Ayguadé, and J. Labarta. Hierarchical Task-Based Programming With StarSs. *Int. J. High Perform. Comput. Appl.*, 23(3):284–299, Aug. 2009.

[106] A. Pothen. Graph Partitioning Algorithms with Applications to Scientific Computing. In *Parallel Numerical Algorithms*, pages 323–368. Springer Netherlands, 1997.

[107] A. Prokopec, P. Bagwell, T. Rompf, and R. Odersky. A generic parallel collection framework. In *Proc. 17th intl. conf. on Parallel Processing*, Euro-Par'11, pages 136–147, 2011.

[108] J. Protic, M. Tomasevic, and V. Milutinovic. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel Distrib. Technol.*, 4(2):63–79, June 1996.

[109] F. A. Rabhi. Abstract machine models for highly parallel computers. chapter Exploiting parallelism in functional languages: a "paradigm-oriented" approach, pages 118–139. Oxford University Press, 1995.

[110] K. H. Randall. Cilk: Efficient Multithreaded Computing, 1998.

[111] L. Rauchwerger and D. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 218–232. ACM, 1995.

[112] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, July 2007.

[113] R. Reiter. Scheduling Parallel Computations. *J. ACM*, 15(4):590–599, oct 1968.

[114] J. V. W. Reynders, P. J. Hinker, J. C. Cummings, S. R. Atlas, S. Banerjee, W. F. Humphrey, S. R. Karmesin, K. Keahey, M. Srikant, and M. Dell Tholburn. POOMA: A Framework for Scientific Simulations of Parallel Architectures. In G. V. Wilson and P. Lu, editors, *Parallel Programming in C++*, chapter 14, pages 547–588. MIT Press, 1996.

[115] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting Dynamic Data Structures on Distributed-Memory Machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[116] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. PPoPP '06, pages 187–197, 2006.

[117] M. L. Scott, M. F. Spear, L. Dalessandro, and V. J. Marathe. Delaunay Triangulation with Transactions and Barriers. In *Proc. 2007 IEEE 10th Intl. Symp. on Workload Characterization*, IISWC '07, pages 107–113, 2007.

[118] C. Smith. *Programming F# 3.0*. O'Reilly Media, 2012.

[119] L. Snyder. The Design and Development of ZPL. In *Proc. 3rd ACM SIGPLAN conf. on History of programming languages*, HOPL III, pages 8–1–8–37, 2007.

[120] M. Steuwer and S. Gorlatch. SkelCL: Enhancing OpenCL for High-Level Programming of Multi-GPU Systems. In V. Malyshkin, editor, *Parallel Computing Technologies*, volume 7979 of *Lecture Notes in Computer Science*, pages 258–272. Springer Berlin Heidelberg, 2013.

[121] M. Steuwer, P. Kegel, and S. Gorlatch. SkelCL - A Portable Skeleton Library for High-Level GPU programming. In *2011 IEEE Intl. Parallel and Distributed Processing Symp. Workshops and Phd Forum (IPDPSW)*, pages 1176 –1182, may 2011.

[122] C. Teijeiro, G. L. Taboada, J. Touriño, B. B. Fraguela, R. Doallo, D. A. Mallón, A. Gómez, J. C. Mouriño, and B. Wibecan. Evaluation of UPC Programmability Using Classroom Studies. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 10:1–10:7, New York, NY, USA, 2009. ACM.

[123] E. Totoni, M. Dikmen, and M. J. Garzarán. Easy, Fast, and Energy-efficient Object Detection on Heterogeneous On-chip Architectures. *ACM Trans. Archit. Code Optim.*, 10(4):45:1–45:25, Dec. 2013.

[124] P. Totoo and H.-W. Loidl. Parallel haskell implementations of the n-body problem. *Concurrency and Computation: Practice and Experience*, 26(4):987–1019, 2014.

[125] P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *J. Funct. Program.*, 8(1):23–60, Jan. 1998.

[126] University of Rome. Dimacs Implementation Challenge. http://www.dis.uniroma1.it/~challenge9/.

[127] T. L. Veldhuizen. Arrays in Blitz++. In *Proc. 2nd Intl. Scientific Computing in Object-Oriented Parallel Environments (ISCOPE98)*, pages 223–230. Springer-Verlag, 1998.

[128] D. A. Wheeler. SLOCCount.

[129] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. In *Workshop on Java for High-Performance Network Computing*, 1998.