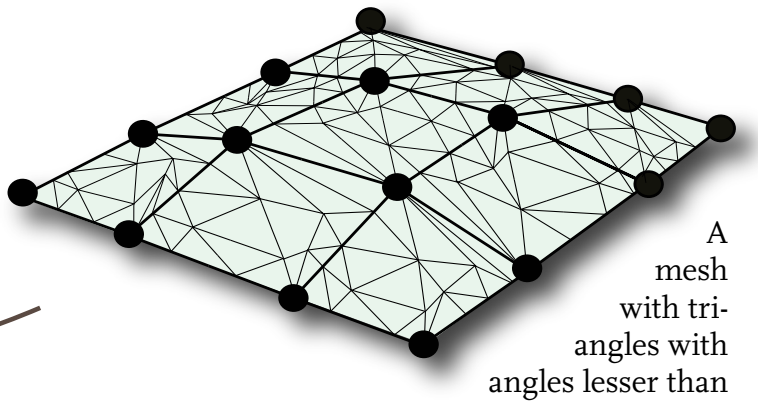
An Algorithm Template for Parallel Irregular Algorithms

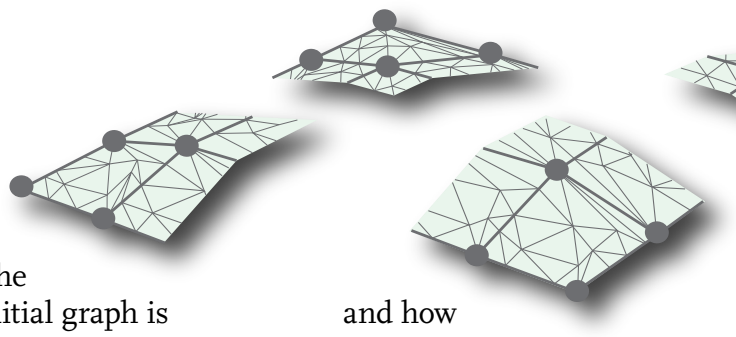
Carlos H. González (*cgonzalezv@udc.es*) and Basilio B. Fraguela (*basilio.fraguela@udc.es*)

Example for Delaunay mesh refinement

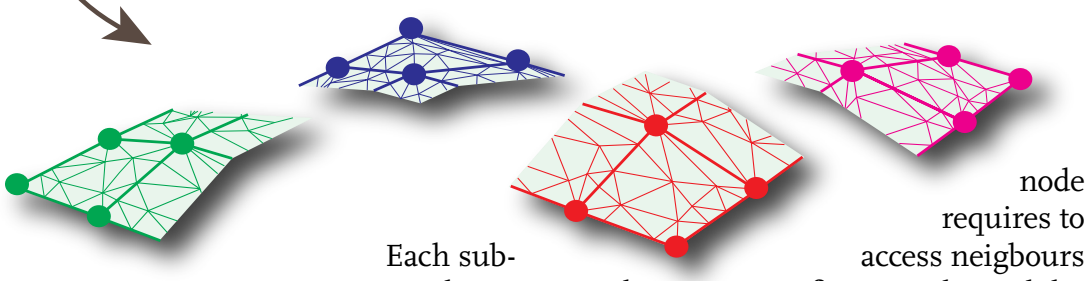
A Delaunay triangulation of a set of points produces a mesh meeting certain conditions. A mesh refinement adds a new restriction: every angle must be greater than 30 degrees.



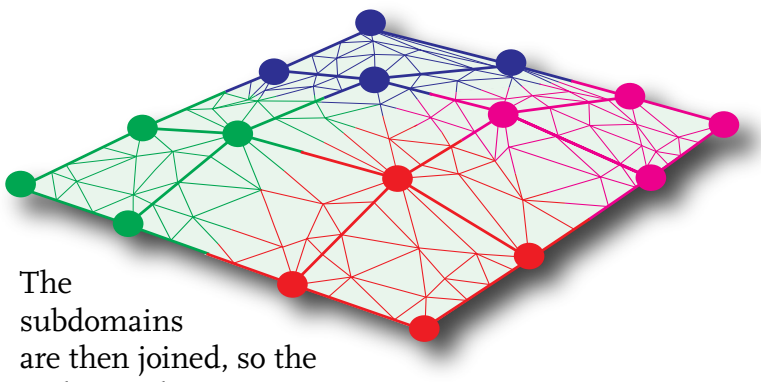
A mesh with triangles with angles lesser than 30 degrees is the input. Each triangle is a node of the graph.



The initial graph is divided using a *Domain*, which defines whether and how to subdivide a graph in smaller subgraphs.



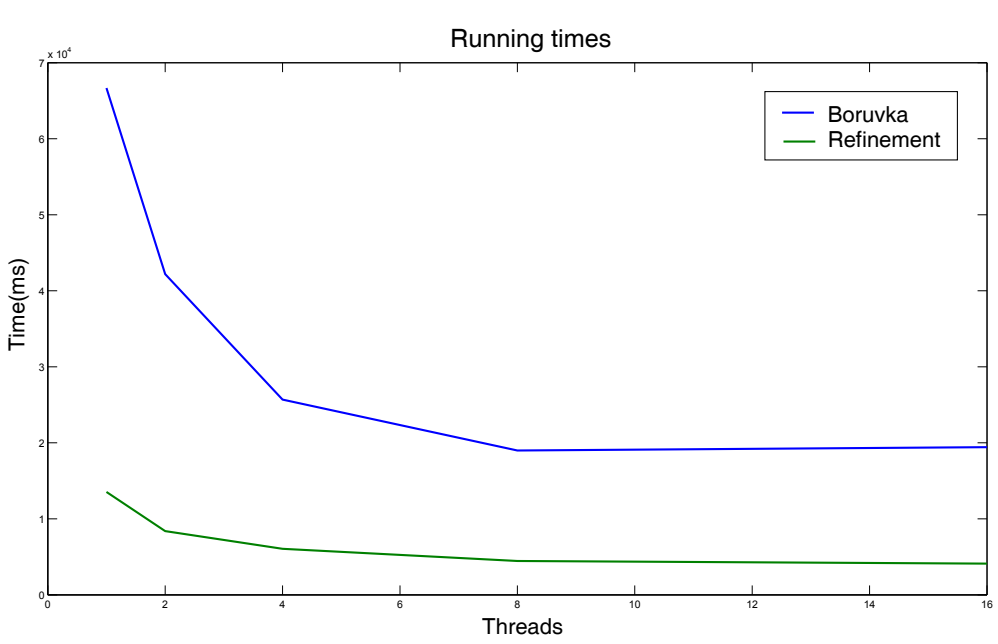
Each subgraph is processed in parallel in a different thread. If processing a node requires to access neigbours from another subdomain, its processing is delayed.



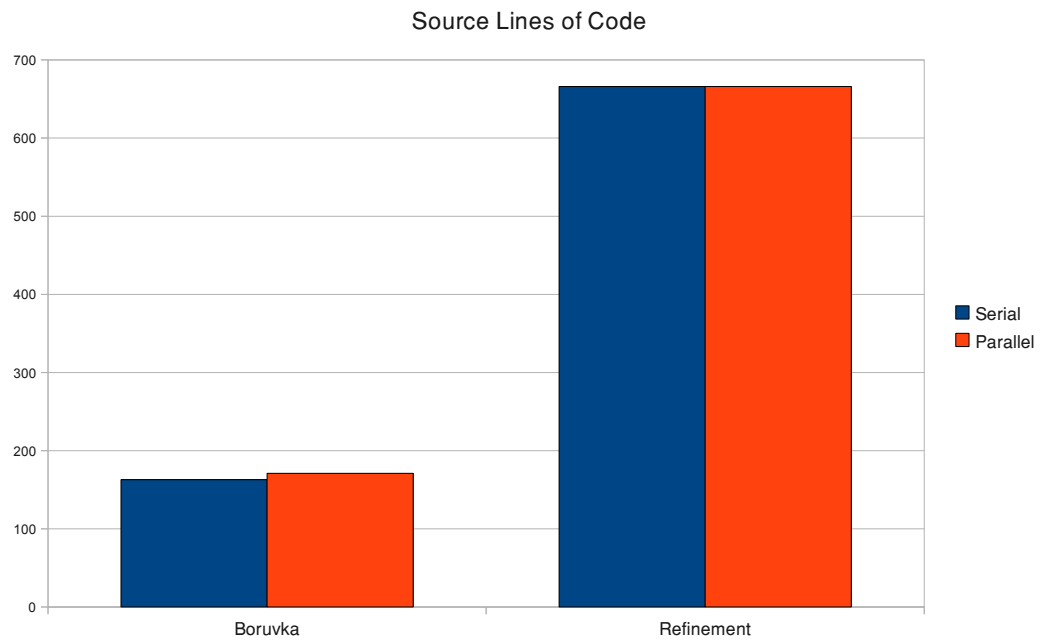
The subdomains are then joined, so the nodes on the seams can be processed, and the final result is obtained.

Experimental results

The template was tested on an HP Integrity rx7640 with different algorithms. Boruvka's algorithm computes a minimum spanning tree of a graph. The figures below show performance metrics.



Running times for Boruvka and Refinement for 1, 2, 4, 8 and 16 cores



Number of Source Lines of Code for serial and parallel versions of the algorithm.

UNIVERSIDADE DA CORUÑA

An Algorithm Template for Parallel Irregular Algorithms

Carlos H. González[*,1],
Basilio B. Fraguela[*,1]

[*] *Dept. de Electrónica e Sistemas, Universidade da Coruña, 15071 A Coruña, Galiza*

ABSTRACT

There is plenty of work done on the area of parallelism regarding regular algorithms and structures, which has led to the development of tools to exploit it. On the other hand, there are other types of algorithms, based on amorphous parallelism, that have not been covered extensively. This work presents a C++ library that abstracts some patterns found in irregular algorithms, easing the development of parallel versions for multicore systems.

KEYWORDS: irregular strucutures, amorphous paralelism, skeleton library

1 Introduction

Tools for parallel programming can be classified in three major groups: languages, compiler directives and libraries. The large majority of these tools are well suited to pallelize regular algorithms, such those found in linear algebra, processing of adjacency matrixes, etc, that use regular data structures, such as arrays which are usually stored adjacent in memory. Thus, the processing work required by these computations is relatively easy to distribute among different cores, based for example on the subset of indices on which each task will operate on each array. Opposed to this regular parallelism, many applications present amorphous data parallelism [KBI+09]. 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.

As a result, parallelizing an irregular algorithm requires more work from the programmer. Each element to process must be manually assigned to a parallel thread. Also, due to the irregular nature of the algorithm, accessing each one of these elements often requires syncronization, normally using mutexes and other syncronization primitives. In this work we present a C++ template library which allows programmers to parallelize almost automatically graph algorithms by relying on the concept of a domain defined on the elements of the graph. In Section 2 the library is briefly described. Section 3 shows a performance evaluation using two different algoritms. Finally, concusions are shown on Section 4.

[1]E-mail: {cgonzalezv, basilio.fraguela}@udc.es

2 Library

This library provides the foundations for developing graph based algorithms, and hides all the complexities explained in the introduction. It is built on top of the Intel Threading Building Blocks [Rei07] library, which provides both a low level API and everal useful parallel skeletons. The core of our library is the `parallel_forall` template function:

`void parallel_forall(Graph * g, Worklist& wl, Domain& dom, Operation& op)`

This function has four arguments: the graph on which it is operating, the initial worklist, a domain definition and the operation to run.

2.1 Graph

The graph is the structure that stores the data required by the algorithm, and is closely related to the `Domain`. A graph comprises nodes and edges mantaining relationships between nodes. Both kinds of element can store data defined by the user of the graph. There are not strong requirements regarding the interface the graph must have to be used with `parallel_forall`. The library provides some classes implementing directed and undirected graphs that the programmer can use.

2.2 Worklist

The worklist is the initial container of elements that are going to be processed. The library provides a definition of Worklist as a C++ standard library `list` for convenience, but any container with the same interface can be used.

2.3 Domain

The domain classifies each element of the graph and the worklist in a different group, depending of its data. The purpose of the domain is to allow the `parallel_forall` function to create several units of work to be run in parallel. The domain is first subdivided recursively, creating several leaf domains. The initial work load is distributed among these subdomains, and then a task is scheduled for each subdomain, which will process the worklist elements belonging to that subdomain. The processing of an element may create new elements that should be processed. If so, they are added to the subdomain task local worklist. When processing an element requires access to neighbours that fall in other subdomain, its processing must be deferred. When the two subdomains of a given domain finish their processing, a task associated to the parent domain is ruan. This task tries to process the elements that coud not be processed in the children. This process happens repetitively until the root of the tree of domains is reached. This root is the initial domain provided by the user, which must cover the whole graph and worklists. This ensures that all the computations are going to be performed.

A class that defines this domain must be provieded by the user, with methods to check whether an element falls inside a domain, to check whether a domain is splittable, and to perform this split. The library includes a domain class that defines a plane, where each element has a vertical and horizontal coordinate.

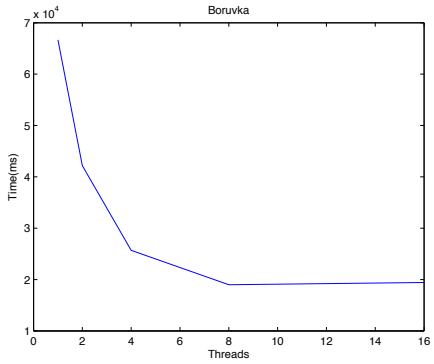


Figure 1: Boruvka running times

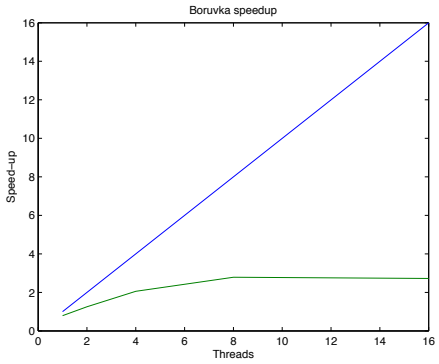


Figure 3: Boruvka speed-ups

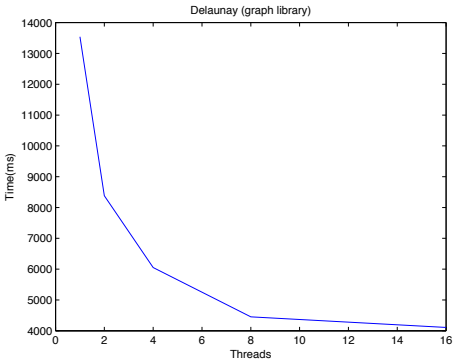


Figure 2: Mesh refinement running times

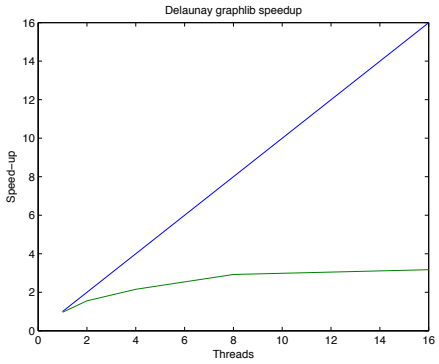


Figure 4: Mesh refinement speed-ups

2.4 Operation

The operation to be done on each element of the worklist. It gets as parameters the current element of the graph to process, the local worklist and the current subdomain. This parameter can be a functor object, a function pointer or a C++0x lambda function. When accessing the neighbours of a node, the operation is responsible for checking whether these neighbours fall outside the current domain.

3 Evaluation

The performance of the library was tested with two benchmarks. One is Boruvka's algorithm, which 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. In the case that there are duplicate edges, only the one with least weight is carried through in the union.

The other implements a mesh refinement of a 2D Delaunay mesh, a triangulation of a set of points with the following property: the circumcircle of any triangle in the mesh must contain no other point from the mesh. A refined Delaunay mesh is a Delaunay mesh with the additional constraint that no triangle have an angle less than 30 degrees.

We implemented a sequential and a parallel version of each benchmark. Parallelizing a program with our library requires just a couple of extra source code lines, for defining the

initial domain and the worklist. The programs were tested on a HP Integrity rx7640 with 16 Itanium Montvale cores, running at 1.6Ghz, and 128 GB of RAM.

The Boruvka algoritm was run with a 2000×2000 mesh-type graph with a domain defined as a plane comprising all the nodes, who had each one a horizontal and vertical coordinate. The plane was divided so each physical thread could have two leaf subdomains assigned. Figure 1 shows the running time for this benchmark using 1, 2, 4, 8 and 16 cores. The speedups with respect to the sequential version are shown in figure 3.

For the mesh refinement benchmark we used an input consisting on the triangulation of a set of 250 000 random points. As in the previous case, the points lied in a 2-dimensional plane, which was the domain used, with two subdivisions per thread too. Figure 2 shows the running times and Figure 4 the speedups.

In both cases, the speedups do not go beyond 3x. This is due to the nature of the algorithms and data structures, which are memory bound. Irregular accesses to memory due to unpredictable indirections are harder to optimize in this stage of the development of the library, and limit the achievable speedup.

4 Conclusion

This work presents a new template library that allows programmers to easily parallelize their programs, so they can make better use of multicore architectures. This library is centered on algorithms on graphs, a very common and general irregular data structures, and relies on the concept of domain to partition work among parallel threads and avoid processing conflicts. A programmer can parallelize highly irregular applications on multicore computers with almost no development cost using our library.

References

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

[Rei07] James Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.