ABSTRACT

FPGA-BASED IMPLEMENTATION OF PARALLEL GRAPH PARTITIONING

by

Mohammad Kharashgeh

Graph partitioning is a very important application that can be found in numerous areas, from finite element methods to data processing and VLSI circuit design. Many algorithms have been developed to solve this problem. Of special interest is multilevel graph partitioning that provides a very efficient solution. This method can also be parallelized and implemented on various multiprocessor architectures. Unfortunately, the target of such implementations is often unavailable high-end multiprocessor systems. Here a parallel version of this method for an in-house developed multiprocessor system is implemented on an FPGA. The system designed provides a cost-effective solution.

The design is based on two Altera soft IP Nios processors. They are synchronized using shared locks. Also, they communicate information by writing messages into buffers. These buffers are also implemented with shared memory.

The design was tested for various graph sizes. The speedup was not attractive for the small graphs but becomes much better as the size of the graph increases. A speedup up to 22% was achieved compared to the single processor design. Larger graphs could yield better speedups. The quality of the partitions produced was also close to the numbers achieved by a single processor. Balance constraints were forced on the partitions and the variations were within 2% of the optimal ones.

FPGA-BASED IMPLEMENTATION OF PARALLEL GRAPH PARTITIONING

by
Mohammad Kharashgeh

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Department of Electrical and Computer Engineering

August 2006

APPROVAL PAGE

FPGA-BASED IMPLEMENTATION OF PARALLEL GRAPH PARTITIONING

Mohammad Kharashgeh

Dr. Sotirios G. Ziavras, Thesis Advisor Professor of Electrical and Computer Engineering, NJIT	Date
Dr. Edwin Hou, Committee Member Associate Professor of Electrical and Computer Engineering, NJIT	Date
Dr. Jie Hu, Committee Member Assistant Professor of Electrical and Computer Engineering, NJIT	Date

BIOGRAPHICAL SKETCH

Author:	Mohammad Kharashgeh
Degree:	Master of Science
Date:	August 2006

Undergraduate and Graduate Education:

- Master of Science in Computer Engineering,
  New Jersey Institute of Technology, Newark, NJ, 2006

- Bachelor of Science in Computer Engineering,
  Jordan University of Science and Technology, Irbid, Jordan, 2003

Major: Computer Engineering

To my parents

ACKNOWLEDGMENT

I would like to thank my thesis advisor, Dr. Sotirios Ziavras, for everything he provided throughout my thesis work. Also, I would like to thank my thesis committee members: Dr. Jie Hu and Dr. Edwin Hou.

Also, I would like to thank my colleagues in the research lab: Xizhen Xu and Zafrul Hasan for their valuable help. And thanks to my friends Emad, Mohammad Al-Shara, Yazan and Ammar for their support.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

GRAPH PARTITIONING

1.1 Introduction

Graph partitioning, or the related topics of mesh partitioning and node tearing, are very important problems in various domains. Applications of graph partitioning include VLSI design, finite element method analysis, scientific simulations, and data processing.

1.1.1 Problem Statement

Given a graph G = (V, E) which consists of a number of weighted vertices (V) and a number of weighted edges (E), our goal is to partition this graph into K sub-graphs (partitions) while minimizing the total cost of the edge cut between these sub-graphs and also keeping the partitions well-balanced. To keep the partitions well-balanced, the graph partitioning algorithm should minimize the variation between different partition weights (the sum of all vertices in a given partition).

1.1.2 Algorithmic Complexity of the Problem

The problem of finding the optimal (best) partitioning of the graph is considered to be NP-complete (can not be solved in polynomial time) [3]. It complexity grows exponentially with the size of the graph [2].

Several heuristics and techniques have been developed to produce suboptimal solutions (local minimum solutions) close to the optimal in an acceptable amount of time. In the next section, a brief description of some of these techniques is given.

1.2 Graph Partitioning Techniques

Many algorithms have been developed to solve the graph partitioning problem targeting various criteria in how to partition a graph. In [1], the authors classify the graph partitioning algorithms into five main categories: geometric techniques, combinatorial techniques, spectral methods, multilevel schemes, and combined schemes.

1.2.1 Geometric Techniques

Geometric algorithms compute a partition of a graph assuming that the node coordinates are available. The idea of this family of algorithms is to try to group nodes that are geometrically near to each other regardless of the connectivity between them. If the coordinates are not available, some methods may be used to devise such coordinates based on graph connectivity. Typically, the geometric methods are very fast but the quality of the partition produced is much less than those produced by other algorithms. So, many trials of this technique may be performed and the one with the best quality may be selected.

Examples of specific geometric methods are: Coordinate nested dissection [1], recursive inertial bisection [1], and space-filling curve techniques [1].

1.2.2 Combinatorial Techniques

Combinatorial methods produce graph partitions based on the adjacency information in the graph. Different heuristics have been developed under this category. One of the most famous algorithms of this type is the one developed by Kernighan and Lin back in 1969 [2]. Their method starts with an arbitrary partition for the graph and then tries to improve the result (i.e., reduce the weight of the edge-cut between partitions) by swapping subsets of nodes between partitions in a way that leads to net reduction in the edge-cut.

The Kernighan/Lin method produces good quality partitions. On the other hand, it is relatively slow and not easy to parallelize as vertex movement may conflict when done concurrently by different processors.

Newer techniques (such as multilevel partitioning) are proven to be superior to the Kernighan/Lin (K/L for short) method. But, actually these newer methods make use of the K/L method or one of its variations in one of their phases (such as the refinement step).

Generally, the K/L algorithm and its variations (such as Fiduccia-Mattheyses) are used in repartitioning. Repartitioning means that a partition is already available and further improvement will be done using a refinement algorithm.

1.2.3 Spectral Methods

Spectral methods work by computing the second eigenvector of the discrete Laplacian of the graph [1]. Spectral methods produce high quality partitions but it is computationally intensive, so it is relatively slow. Several improvements have been made to the basic algorithm to make it faster.

1.2.4 Multilevel Techniques

Multilevel algorithms partition the graph in three phases. In the first phase, namely coarsening, the input graph is shrunk and coarsened into a related smaller graph. This process continues until the coarsest graph is of a manageable size. In the second phase, namely initial partitioning, any method for graph partitioning could be used to get good partitions for the coarse graph. The last phase, namely uncoarsening, is projecting the coarser graph back onto the finer one and at the same time refines the partition produced at the coarser level [3].

Multilevel algorithms are fast and produce graph partitions of high quality. These algorithms are the main focus of this document and will be discussed in more detail in later sections.

1.2.5 Combined Schemes

Different partitioning techniques could be grouped together to produce better partitioning schemes. For example, a fast but low quality partitioning is first produced by geometric methods, then the solution is improved by using another method like the K/L algorithm [1].

Also, a multilevel scheme can use a refinement algorithm based on the K/L algorithm in the uncoarsening phase [8]. Some algorithms are available that adapt the multilevel approach and use spectral methods to compute the initial partitioning of the graph.

1.3 Multilevel Graph Partitioning

Multilevel techniques provide very efficient solutions for the graph partitioning problem. The quality of the partitions produced is high and the run time is small compared to other techniques [1]. In addition, the multilevel algorithms can be parallelized efficiently. The above are the reasons why we chose to implement this method on our multiprocessor system [see the next chapter for the design details].

1.3.1 Introduction

As stated before, the multilevel technique includes three phases: Coarsening the graph several times until it is of a manageable size, then partitioning the coarse graph, and finally projecting the partitioned graph back to the finer ones step by step while performing refinement (using a K/L variation) at each step.

We illustrate the idea of multilevel techniques and their efficiency as follows. We start by looking back at the K/L algorithm. As stated in [2], an initial partitioning is chosen (most probably randomly) and then it is improved iteratively by swapping subsets of the partitions in a way that improves the partitions. On the other hand, multilevel algorithms start refinement with a very good initial partitioning in the second phase (which greatly affects the quality); they also carry out much faster in refinement because at coarser levels moving a vertex means moving a group vertices at finer levels.

The following subsections illustrate the three phases of multilevel graph partitioning in greater detail.

1.3.2 Graph Coarsening

The input in this phase is the original graph to be partitioned and the output is a related graph of a much smaller size. To reach the intended coarse graph, successive application of a graph matching algorithm is used.

A precise definition of graph matching is as follows: Find largest-size set of edges S from E, where E represents the set of all edges, such that each vertex in V, where V represents the set of all vertices, is incident to at most one edge in S. Although the above definition is correct, it is a little bit not clear in our case. For simplicity, we consider graph matching as finding the maximal number of groupings between two neighbor vertices where each vertex can participate just once (matched with only one neighbor vertex).

For example, assume the input graph to be partitioned in figure 1.1.

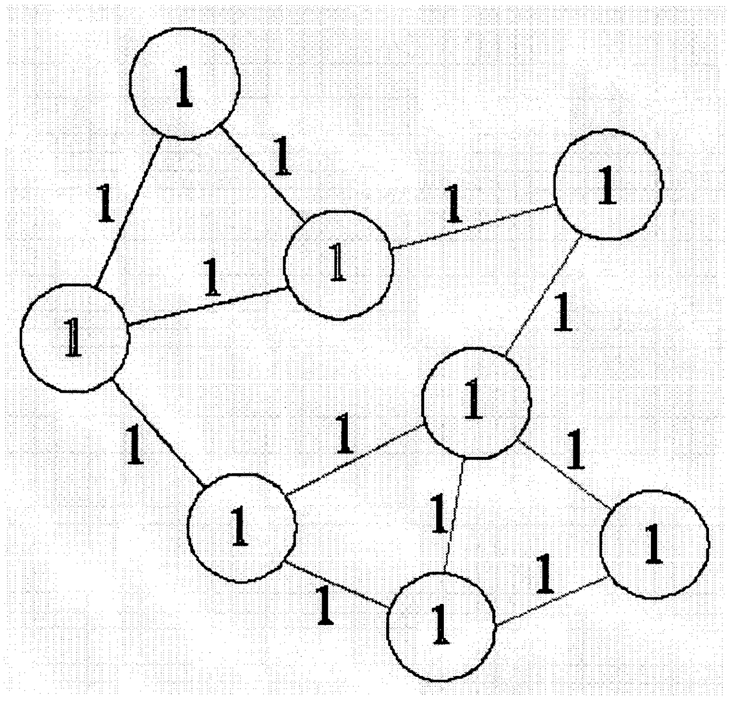


Figure 1.1 Example of an input graph.

The above graph has eight vertices and eleven edges. All the vertices and edges have the weight of one (as indicated).

Now we will match every two neighbors. Many choices are available for doing this and one of them is illustrated in figure 1.2

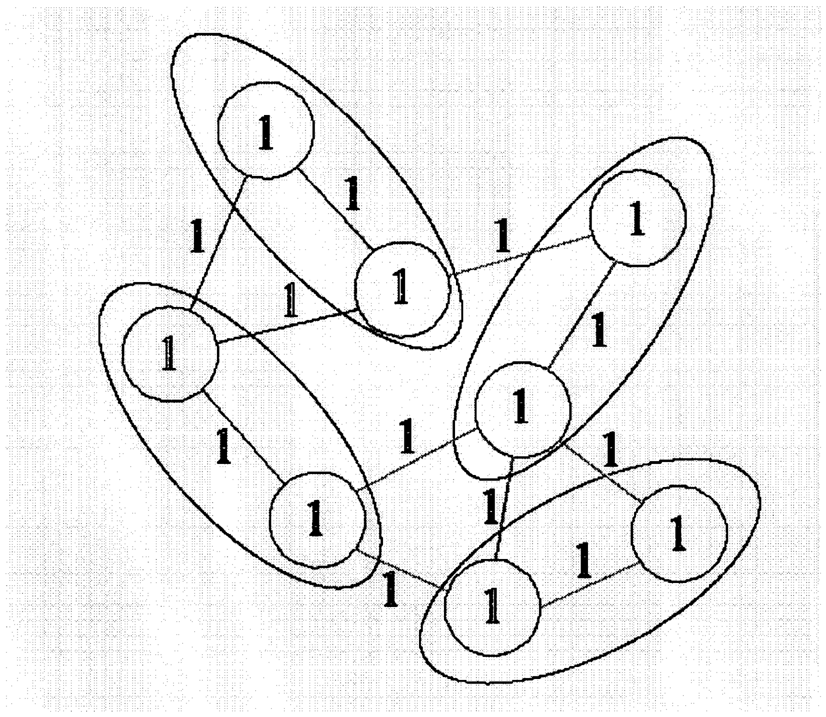


Figure 1.2 Sample matching of a graph.

The next step after matching is to construct the coarser graph. This is done by merging the matched vertices and their weights. The resultant coarser graph is shown in figure 1.3:

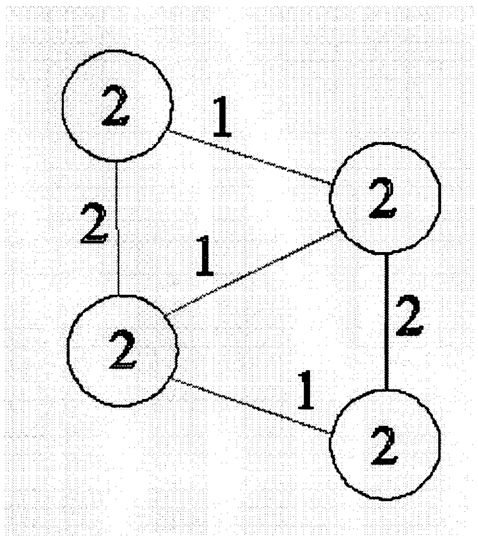


Figure 1.3 Coarser graph.

Different methods are suggested for graph matching. In [3], a number of methods are described. For example:

- Random matching: a vertex can be matched with any neighbor vertex (not matched yet).
- Heavy-edge matching: a vertex is matched with the neighbor vertex corresponding to the heaviest edge. This scheme helps in producing higher quality partitions.

1.3.3 Initial Partitioning

In this phase the coarser graph can be partitioned using many different algorithms [8]. Figure 1.4 illustrates how the above coarser graph can be partitioned into two parts. The best edge-cut at this level is 3.

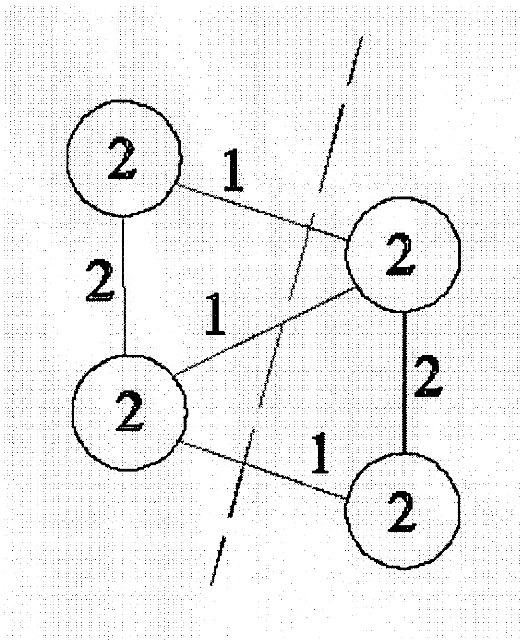


Figure 1.4 Initial partitioning of coarser graph (edge-cut is 3).

In this project, we use a specific initial partitioning algorithm called Grow Bisection [8]. This algorithm starts with a random vertex as the initial partition. Then, it augments this small partition by adding vertices to it in a breadth-first fashion. This continues until the partition reaches the needed specific total weight.

Since this procedure is sensitive to the choice of the starting vertex, several trials can be done and the one with the smallest edge-cut (best quality partitioning) may be chosen [8].

1.3.4 Uncoarsening and Refinement

The first step in this phase is to project the current partition back to the finer graph at the next level. Figure 1.5 illustrates this step. As you can see, the edge-cut is the same (3) as the edge-cut of the previous level.

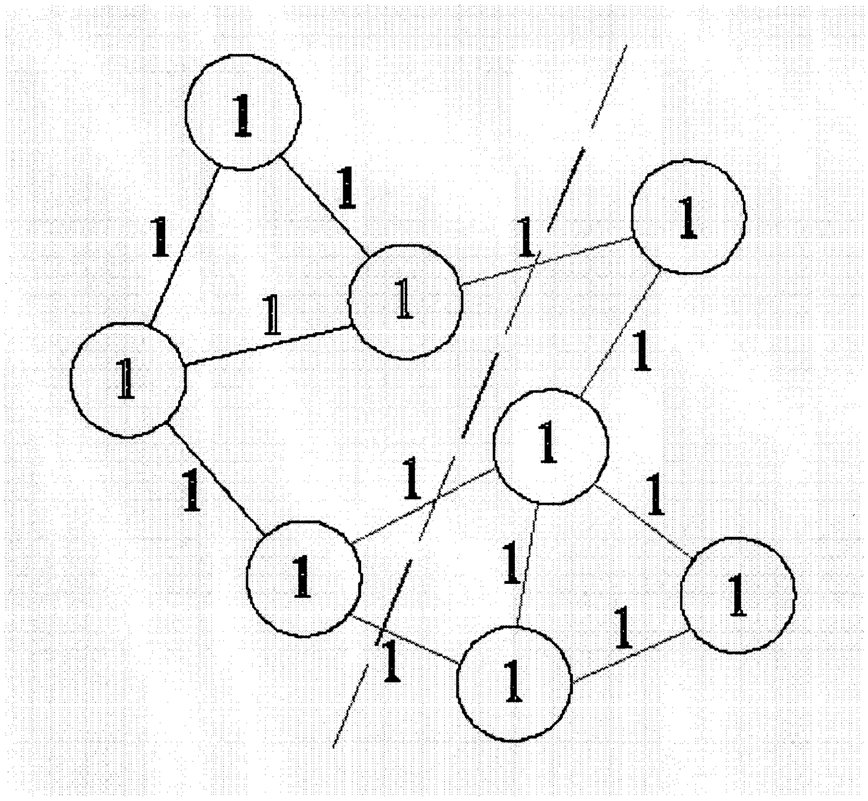


Figure 1.5 Projecting a partition back to the next finer level.

One can notice that although the edge-cut between partitions at the coarser level was minimum, it is not the case at the finer level (a better edge-cut is achievable). Here comes the importance of refinement.

Graph refinement is performed using the K/L algorithm or one of its variations. As a result, two vertices in our case could be swapped to produce higher quality partitions (the edge-cut is just 2) as illustrated in figure 1.6

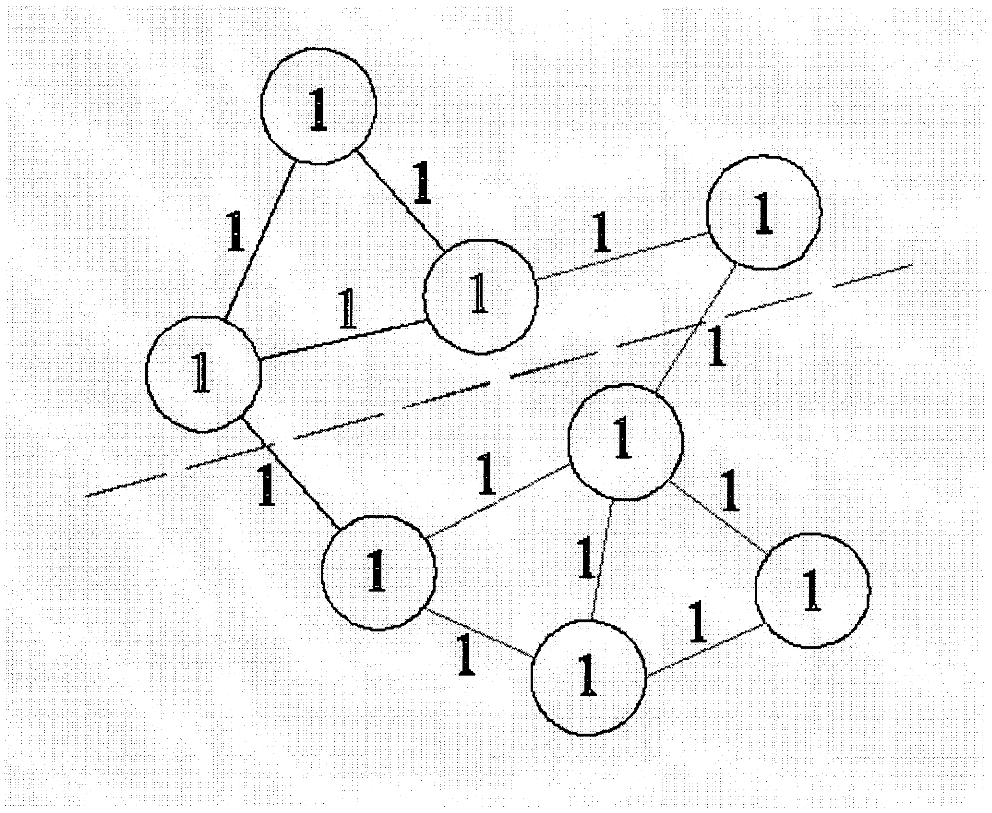


Figure 1.6 Refinement improves the partitioning quality

A specific refinement algorithm, Greedy Refinement[3], is implemented in this project. Greedy Refinement (GR for short) works by computing the gain if a vertex is moved from one partition to another. Under this scheme, either vertices with positive gain are moved or vertices of zero gain when their movement improves the balance between partitions.

1.4 Parallel Multilevel Graph Partitioning

Many efforts have been made to parallelize multilevel graph partitioning [9], [10], [11], and [12]. All algorithms keep the single processor spirit by using the same three phases. But additional computations and unavoidable overheads are added to the basic process.

For the coarsening phase, conflicts between processors may arise as vertices on different processors may try to match the same vertex at the same time. So, [11] suggests performing an extra step of graph coloring before coarsening. Graph coloring assigns colors to the vertices so that no two adjacent vertices have the same color. Graph coloring can be used in matching to reduce conflicts by matching vertices of one color at a time. Also, coloring happens to be essential in the refinement process.

For the initial partitioning phase, since the run time of this phase is small relative to the other phases, this phase could still be performed on one of the processors. So, after the end of the coarsening phase the coarser graph is moved to one of the processors in order to be partitioned there.

In the uncoarsening phase, conflicts may also arise in concurrent vertex movements as there are cases in which moving a vertex to another partition by a processor may improve the quality but not when another vertex is moved by another processor at the same time. So, graph coloring is used in the refinement phase to remove conflicts. This is done by exclusively considering for movement vertices of one color at a time.

Synchronization is needed between processors in the matching and refinement steps. This is done as follows: every processor works on its subset of vertices with the current color, then, it waits for all the other processors to be done too with this iteration.

This is important to reduce conflicts as stated before. At the end of the iteration important information is communicated between processors.

Interface vertices (vertices on the boundaries between processors) require special attention in the parallel formulation. Some interface vertex information should be duplicated on both end processors. After each iteration, updated information about the interface vertices is sent to the other boundary processor. This information includes current color, match, and partition of this interface vertex.

To summarize the parallel graph partitioning process: First, the input graph is distributed evenly among the processors in a chosen way (in [10], it is shown that the choice of this distribution does not affect the output quality). Then, the graph is colored. After that, the coloring is used to aid in matching and, hence, constructing the coarser graph. Then the coarser graph is manipulated by a single processor in order to produce the initial partitions. The resultant partitions are sent to the corresponding processors. Next, uncoarsening and refinement is performed – making use of graph coloring.

The goal of parallel graph partitioning may be different from one design to another. For some cases, there is no relation between the graph partitioning output and the underlying multiprocessor system; i.e. the goal is to specify a partition number for every vertex. On the other hand, the ultimate goal of other graph partitioning schemes is to have the partitions assigned at the end to specific processors. For such case it is usual to have the number of intended partitions equal to the number of processors.

The latter case identifies two approaches for solving the problem. The first one 'actually' moves vertices between processors doing refinement. The second approach uses the idea of virtual immigration. Virtual immigration does not actually move vertices, but just changes the partition number for that vertex when intended for movement. The second approach is much faster but needs an extra step at the end to redistribute the vertices to the correct processors.

1.5 Motivation and Objectives

Graph partitioning is a very important problem. Fast graph partitioning is critical in many applications and domains. Multilevel algorithms provide efficient solutions. They can also be parallelized on multiprocessor systems. Unfortunately, these implementations often target unavailable high-end multiprocessor systems. Our goal is to designing an FPGA-based multiprocessor system to perform parallel graph partitioning. Thus, we want to provide a cost-effective solution to this important problem.

CHAPTER 2

FPGA TECHNOLOGY

2.1 Introduction

FPGA stands for Field-Programmable Gate Array. As stated in [4]: "A field programmable gate array (FPGA) is a large scale integrated circuit that can be programmed after it is manufactured rather than being limited to a predetermined, unchangeable hardware function."

FPGAs are considered the main component for a family of architectures called reconfigurable architectures. Before FPGAs, there were two main approaches for the execution of algorithms [5]: hardwired technology (such as ASICs) or software-based approach using microprocessors.

ASICs (Application Specific Integrated Circuits) are designed to perform exclusively a specific job. Once ASICs are manufactured for a specific application, they can not be changed. So, ASICs lack flexibility in functionality. On the other hand, ASICs have the advantage of being extremely fast in performing the specific job they are designed for.

The other approach, software-based microprocessors, is extremely flexible and different applications could be performed by the same microprocessor. Some overhead in performance is encountered to maintain this flexibility.

FPGAs are somehow a trade-off between the two approaches. Allowing the designer to have his/her own custom hardware logic to perform critical functionality and at the same can incorporate microprocessor(s) in the design. Added to this is the ability to dynamically reconfigure the system, if needed.

2.2 FPGA Resources

A typical FPGA contains the following resources [4]:

- Logic elements: These are arrays of basic logic resources used by a process called mapping to perform the desired logic function.
- Lookup tables (LUTs): Basically, they are ROM-style hardware that can be used to implement any combinational function. LUTs as well are usually the building blocks of logic elements.
- Memory resources: Most FPGAs contain on-chip memory resources such as SRAM that could be used as local memory.
- Routing resources: A considerable portion of the FPGA chip is allocated for routing purposes. When configuring the FPGA, these routing resources will be used to connect different hardware to produce the desired functionality.
- Configurable I/O: Typical FPGAs have programmable I/O features. Many options are available to the designer to program pins and interfaces to suite his/her needs.

2.3 FPGA Types

According to [6], there are two main styles or types of FPGA technology: SRAM-based and antifuse-based:

- SRAM-based: FPGAs of this type hold their configurations in static memory. This type has the advantage of easy programming.The SRAM contents can be changed dynamically through reprogramming to reconfigure the FPGA. Most FPGAs are of this category.

- Antifuse-based: To program FPGAs of this type, a voltage is applied across the wanted antifuses. Each antifuse is programmed separately.

2.4 IP Cores

IP stands for Intellectual Property. One of the main advantages of designing FPGA-based systems is the availability of IP cores. These are ready-to-use components that can be easily instantiated in the design.

Different vendor libraries are available for IP cores. These cores could be as simple as an adder and as complex as a fully featured microprocessor. Several IP cores are also available for many peripheral devices.

2.5 The Nios II Development Board

Most FPGA chips come as part of a development board. This board will include resources other than the FPGA chip, such as on-board off-chip memory, and I/O ports and interfaces to aid in the communication between the FPGA-based system and other devices.

The multiprocessor graph partitioning system described in this document [see next chapters] was implemented on the Altera Nios II Development Board – Cyclone II edition [7].

The main component of this board is the Altera Cyclone II FPGA. This FPGA has 33,216 logic elements. If the economical Nios II processor IP is used then the number of logic elements in this FPGA is enough to instantiate around fifteen Nios II microprocessors.

Also, this FPGA contains 483,840 bits of on-chip memory, for a total of about 60KB; they could be used as local memory. In addition, the following are available on this board:

- 2 Mbytes of synchronous SRAM.
- 32 Mbytes of DDR SDRAM.
- 16 Mbytes of Flash memory.
- RS 232 serial connector.
- JTAG connector for FPGA configuration purposes.
- Ethernet connector.
- Oscillator to serve as clock input.
- Other devices (such as LEDs and push buttons) .

CHAPTER 3

HARDWARE ARCHITECTURE

3.1 Introduction

In this chapter we describe the designed multiprocessor architecture and its various components for the graph partitioning problem.

It is a two-processor system that uses messages for interprocessor communication and shared variables to synchronize. Figure 3.1 shows a high level diagram of the system.

Actually, different memory and communication configurations were used. Each choice has its pros and cons. Initially, we used a communication scheme based on Altera components and library for mailboxes described in detail in section 3.3. This approach has the advantage of easy programming (library support and communication constructs provided), but suffers from high delay in sending and receiving data.

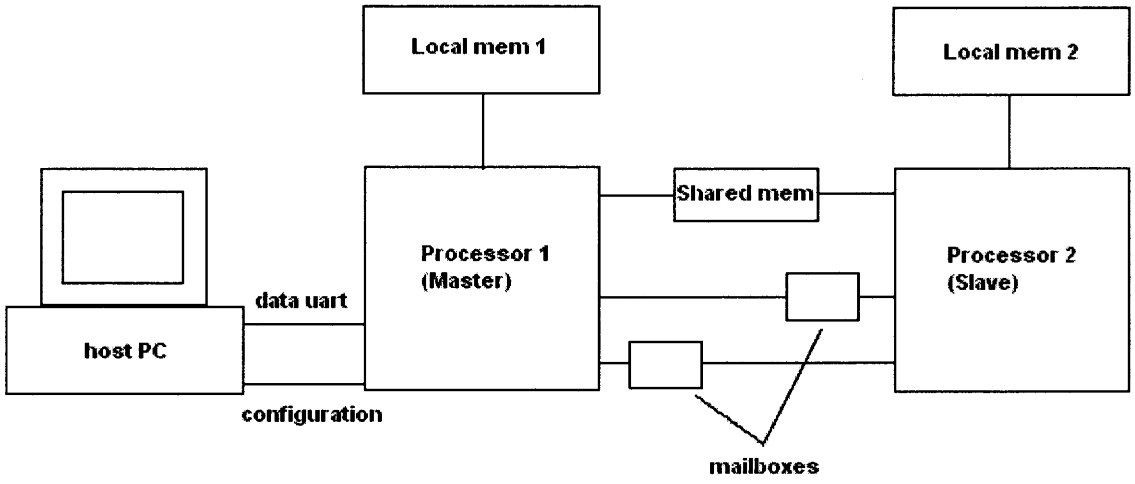


Figure 3.1 High level diagram of the two-processor system.

Another communication approach (much faster and eventually used to get our results) is based on sending and receiving messages by writing them in a specified location of local shared memory. The programmer is responsible of reading the recent data and not overwriting old data based on synchronization.

Mainly two memory configurations were used. The first one uses local memory for both the graph data and message buffers. This configuration gives the fastest results, but unfortunately, local memory is limited and can not be used to test larger graph.

The second memory configuration (The performance chapter uses results using this configuration) uses static RAM to hold graph data and local memory for message buffers. This approach is slower, but provides more space.

3.2 Components Used

The following components were used:

- Two Nios II microprocessors. Nios II processors are 32-bit RISC processors. They are implemented with instruction caches of size 2KB and support for branch prediction.
- Local memories (mem1 and mem2). These memories use on-chip memory bits. The size of each is 16KB.
- Shared memory. It also uses on-chip memory bits. It holds shared variables (locks) for synchronization purposes.
- Two mailboxes. These are ready to use components that are used to communicate messages between processors. Each mailbox is based on

1KB of shared memory to hold messages in a FIFO manner. Each message is deleted once it is read.

- 2MB of off-chip static RAM. This memory is used in one of the configurations as the program and data memory and its address space is partitioned between the two processors.
- UART connection. This serial connection is used to communicate data with the host PC. Its baud rate is 115200 bps.
- JTAG connection. This connection is used to download FPGA configuration to the chip.

3.3 Mailboxes Interprocessor Communication

The two processors communicate with each other using mailboxes. Any processor can write and read from a mailbox (if defined as shared by the design tool)). In our case, we use these mailboxes as one –way devices. This means that one of them for processor 1 to write messages for processor 2, the latter can just read from this mailbox. The situation is the opposite for the other mailbox.

Both of these FIFO mailboxes need a buffer to store messages. These buffers are simply local memory. Each one has 1KB of buffer space; when fully no messages are accepted. Also messages are automatically deleted when read.

Supporting software library is used to access these mailboxes. The functionality such as reading and writing to mailboxes is greatly abstracted and simplified by using the Altera software library.

3.4 Synchronization

The two processors are synchronized using shared locks. Both processors need to be sure that the other one is done with the current software iteration before proceeding to the next iteration. The chosen for synchronization is as follow:

- When processor 2 reaches a synchronization point, it sets the lock and waits for processor 1 to reset it.
- When processor 1 reaches a synchronization point, it waits until the lock is set (by processor 2) and then resets it.

So the execution will continue as soon as both processors reach the synchronization point.

3.5 Communication with the host PC

Two different types of communication are employed between the FPGA board and the host PC:

- The first communication is through the JTAG serial line; by this communication, the host PC downloads the configuration of the FPGA onto the board.
- The other type of communication is through the RS-232 UART interface. By this, the host PC sends the input graph to the FPGA. Library support is used on either side to access the serial line. The above communication is done in the character mode. It is then transformed into integers by the receiving side.

CHAPTER 4

SOFTWARE DESIGN

4.1 Introduction

In this chapter, more details about the graph partitioning software will be given. First, reading the graph from a file and constructing the data structures for the graph will be discussed. Then we will describe the graph C structure used in details. After that, each step of the graph partitioning algorithm will be illustrated.

4.2 Reading a Graph

Two methods of dealing with input graphs are used; one is reading it from a graph file and the other is generating it using a generator program (Appendices A and B provide more details). The ultimate output of both cases should be the same – four adjacency arrays also involving weights. For the first method, we get these arrays by re-using the "ReadGraph" function found in the METIS graph partitioning software [15]. For the other method, these arrays are generated directly.

As stated above, four arrays are used to represent a graph. This is done following the Compressed Storage format (CSR) [15]. The four arrays are:

- adjncy[2*M]: contains the neighbors of vertex 1, then neighbors of vertex 2, and so on. Here M is the total number of edges, and we need 2*M because every edge is listed twice (once by each of the neighbors it connects).

- xadj[N+1]: contains the starting and ending index of the adjncy array for the neighbors of the first vertex then the second and so on. N is the number of vertices.
- adjwgt[2*M]: contains the weight of the corresponding edge found in the adjncy array.
- vwgt[N]: contains the weight of every vertex (in order).

For example, a sample graph is shown in Figure 4.1. No vertex or edge weights are specified for this sample graph. If weights are not specified in the input file, they will have the default value of 1.

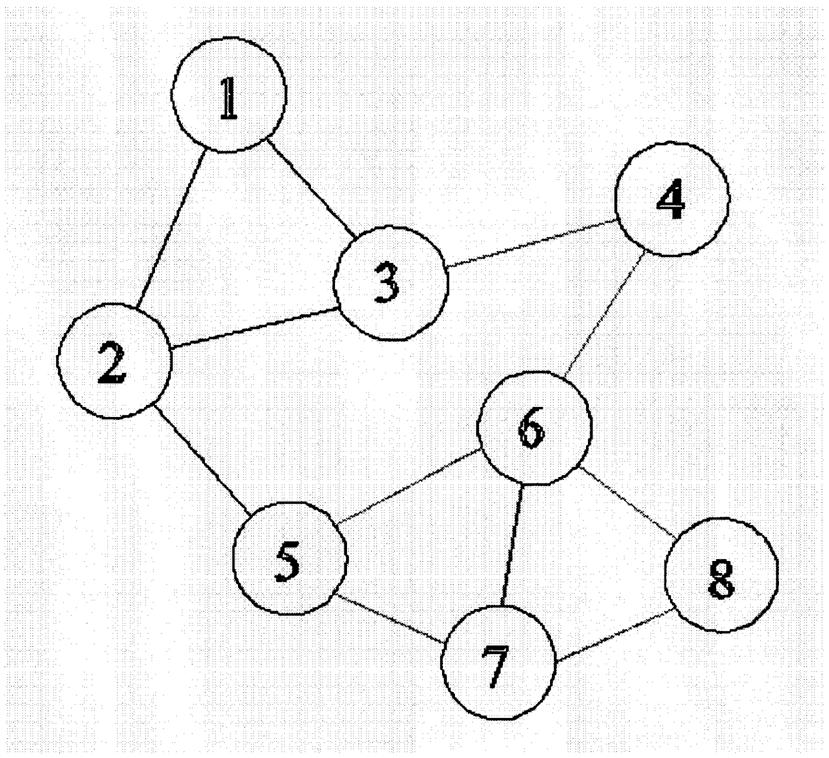


Figure 4.1 Sample input graph

The corresponding CSR format arrays for the above graph would be:

adjncy[22] = {2, 3, 1, 3, 5, 1, 2, 4, 3, 6, 2, 6, 7, 4, 5, 7, 8, 5, 6, 8, 6, 7}

xadj[9] = {0, 2, 5, 8, 10, 13, 17, 20, 22}

Extra spaces in the xadj array were used to illustrate its relation to the adjncy array. The relationship is as follows: For the first vertex, its neighbors are listed in adjncy array starting with the index value of 0 to the index value of 2 (but adjncy[2] is not included). For the next vertex, its neighbors start at index 2 up to index 5, and so on. For the two weight arrays, they will have the default value of 1 for every vertex and edge.

4.3 Graph Partitioning Phases

Figure 4.2 shows a general flow graph for the partitioning process.

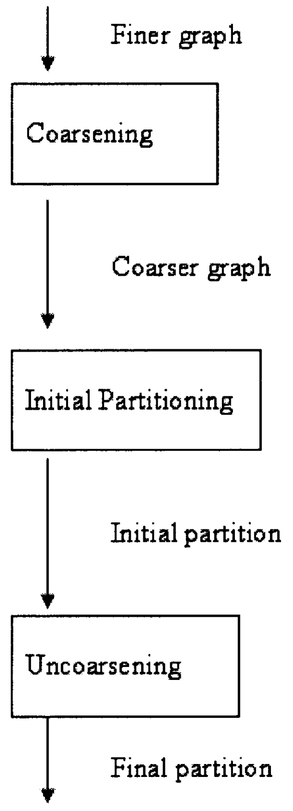
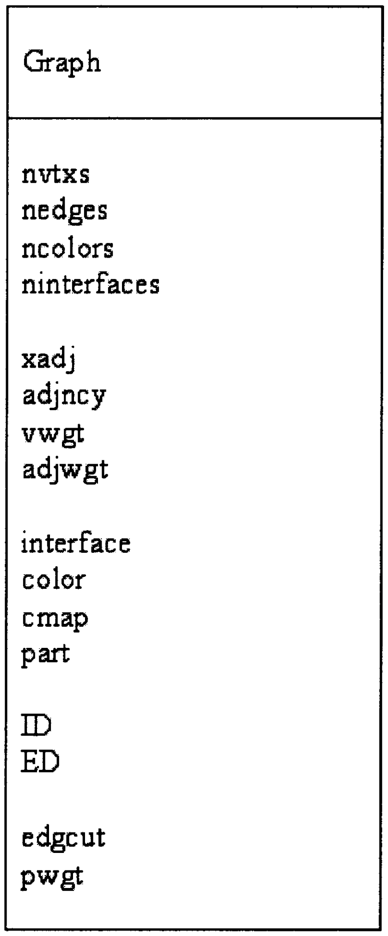


Figure 4.2 Graph partitioning phases

The above diagram is a simplified view of the partitioning process, as most of the phases contain sub-phases. Also, in parallel graph partitioning, an extra step of graph coloring at the beginning is introduced. Software development issues for all of these phases will be described in detail in the following sections.

Figure 4.3 shows the Graph structure used. The first four variables represent the number of vertices, edges, colors, and interface vertices respectively. The third set is four arrays of size 'nvtxs' and specify properties of each vertex (color, partition …). The ID and ED are arrays that contain the internal and external degree of each vertex. 'pwgt' is an array of size of the number of partitions and hold the total weight for each.



Graph
nvtxs nedges ncolors ninterfaces xadj adjncy vwgt adjwgt interface color cmap part ID ED edgcut pwgt

Figure 4.3 Graph structure.

4.3.1 Pre-Processing Phases

Several steps should be done before starting coarsening the graph in parallel; these are: Initial distribution, interface calculations, and vertex coloring.

In the initial distribution sub-phase, the graph is linearly distributed among processors by the master processors. Linear means that if we have two processors, for example, to participate in graph partitioning, the first processor will take the first half of the graph (first have of 'adjncy' and the corresponding information) and the other processor will take the other half. In some cases when there is a variation of processing power between available processors, the more powerful side will take larger part of the graph. In our case, we just use symmetric graph distribution.

It is important to note that the initial distribution phase is completely different from the initial partitioning phase (after coarsening) in the multilevel graph partitioning approach. In some other partitioning approaches, the two phases may mean the same (as in the K/L approach described in chapter 1). But in multilevel graph partitioning, the initial partitioning actually assigns partitions to vertices while initial distribution is just a way to distribute the processing load between participating processors.

In the interface calculations sub-phase, each processor considers all of its vertices and checks if it has neighbors from other processors' domains. If so it sets the interface value of that vertex to 'true', else it will be false.

In the coloring phase, a variation of Luby's algorithm [18] is used. This variation is suggested in [11]. Each processor tries to find a maximal independent set for a given color (without having two neighbor vertices of the same color) then they synchronize and exchange information about the newly assigned colors before starting the next iteration.

Coloring the whole graph is time-expensive. Actually, the algorithm becomes slower for successive colors. So, as suggested in [11] , no need to color the whole graph and coloring a good percentage of it (e.g., 80%) will produce comparable quality measures with much less time.

4.3.2 Graph Coarsening

In this phase, we perform three main steps; graph matching, coarse vertex mapping, and constructing coarser graph.

Many matching conflicts (conflicts happen when a vertex matches another one which at the same time is matched to a third) are avoided by matching the graph in parallel using the computed coloring of the graph. We perform matching by considering vertices of the same color to be matched in one iteration. Vertices of no color (color variable equals -1) can not participate in the matching themselves, but still they can be matched by other colored vertices.

Since no two neighbor vertices can have the same color, interface vertices from different processor domains can not match at the same time. This greatly reduces the possible conflicts in matching.

One conflict scenario is still possible. Figure 4.4 shows an example of that case. Here vertices 2 (from processor 1 domain) and 4 (from processor 2 domain) have the same color, and so, can participate in the current iteration of graph matching. Both of them found vertex 3 unmatched and decided to match. The match variable of vertex 3 may have an unknown value (either 2 or 4). In our case, we give priority to local vertices and it will have value of 2. To allow vertex 4 adjust its wrong value, after each matching iteration the processors synchronize and they update match values. Then every vertex reads the match variable of the vertex it matched. If it is the same as its number, it will proceed normally. Other than that it will set its match variable to -1 to indicate that it is not matched. At the end of graph matching, every vertex should have a vertex matched to it or it will be matched to itself.

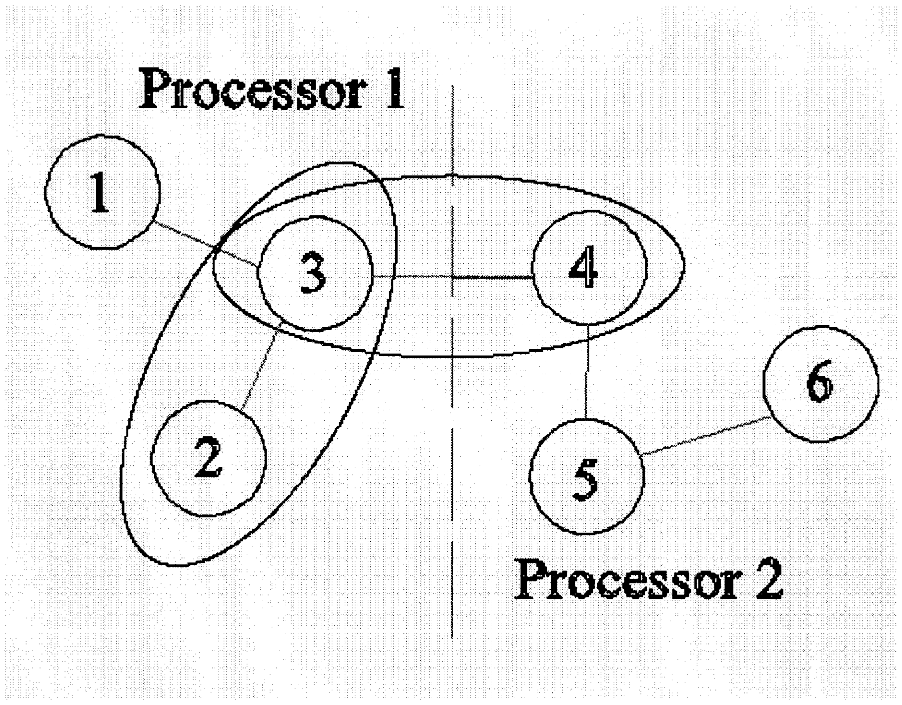


Figure 4.4 Matching conflict.

The next step after matching is generating the coarse map. In this step every vertex is mapped ('cmap' variable assigned a value) to another vertex at the coarser level. Of course matched vertices in the finer level should be mapped to the same vertex at the coarser level.

The last step in coarsening is constructing the actual coarser graph based on matching and coarse map. This is the most complicated step in terms of code size.

This step starts by changing the vertex numbers from the previous level to the new vertex numbers at the coarser level using the coarse map generated previously. After that, the processors synchronize and send adjacency information about interface vertices. Then the adjacency lists of the matched vertices are grouped together to form the adjacency information of the new coarser graph. At the same time weights of the new vertices and edges are updated.

4.3.3 Initial Partitioning

This step is relatively fast and can be done serially without affecting performance. So, the coarse graph pieces is grouped from different processors and given to the master processor which performs the partitioning and then sends back the result to the different processors.

In this phase, any partitioning algorithm can be used to partition the coarse 'relatively small' graph. In our case we use Grow Bisection algorithm [11]. For example, if we want to partition the graph into two parts, we will start by having all vertices in partition number 1. After that, we select a vertex and make it the seed for partition 2. Then we augment partition 2 by adding vertices using breadth first approach starting from the seed.

4.3.4 Uncoarsening

Two steps are done in this phase; projecting the partition back to the original graph, and refining the partition at each level to improve the quality of the partition (by reducing the edge-cut).

In the first step, every vertex at the finer level is assigned the partition of the corresponding vertex at the coarser level.

The second step, namely refinement, is the most important step in the algorithm. In this step the quality of the initial partition is improved iteratively. The refinement algorithm used in this step is Greedy Refinement (GR) described in [11].

The GR algorithm is based on the previously computed coloring of the graph. It performs vertex movement between partitions to reduce edge cut iteratively; one iteration for every color.

An iteration starts by calculating the internal degree and external degree for each vertex, and storing these values in the ID and ED arrays respectively. Internal degree of a vertex can be found by summing the weights of the edges connecting it with vertices having the same partition. On the other hand, external degree of a vertex is the sum of the edge weights connecting it to vertices belonging to another partition.

A vertex will be moved to another partition only in these two cases:

- If the value ID – ED is greater than zero for a vertex and moving that vertex to the other partition will not make that partition total weight exceeds the maximum imposed by balance constraints.
- If the ID – ED = 0, and moving the vertex will improve the balance between partitions.

At the end of the iteration, the processors are synchronized. Then they communicate to calculate the new values of the edge cut and the weights of the partitions after vertex movements.

CHAPTER 5
PERFORMANCE MEASURES

5.1 Introduction

In this chapter, the results for graph partitioning using our two-processor system are presented. The test graphs used are regular grid structures (NxN graphs). Although irregular graphs better represent practical problems, we used the mentioned regular graphs to study the relation between the size of the graph and the speedup gained. We used for the tests three different sizes: from 10x10 (100 vertices) to 40x40 (1600 vertices).

Four aspects of graph partitioning will be considered, namely speedup as compared to a single processor system, quality of partitions, balance among partitions, and memory usage. These measures are explained in the following sections.

5.2 Speedup

The parallel two-processor implementation is not a direct shift from the single processor. This means that the code for the single processor is not just duplicated to the two processors with minimal effort and overhead. Actually, the multiprocessor implementation has many aspects irrelevant to the single processor one. Graph coloring is used to avoid conflicts in the matching and refinement phases, interface vertices require extra computing, and other topics like vertex ownership and numbering have to be treated.

Figure 5.1 illustrates the time needed for partitioning the test graphs on the single and multiprocessor designs.

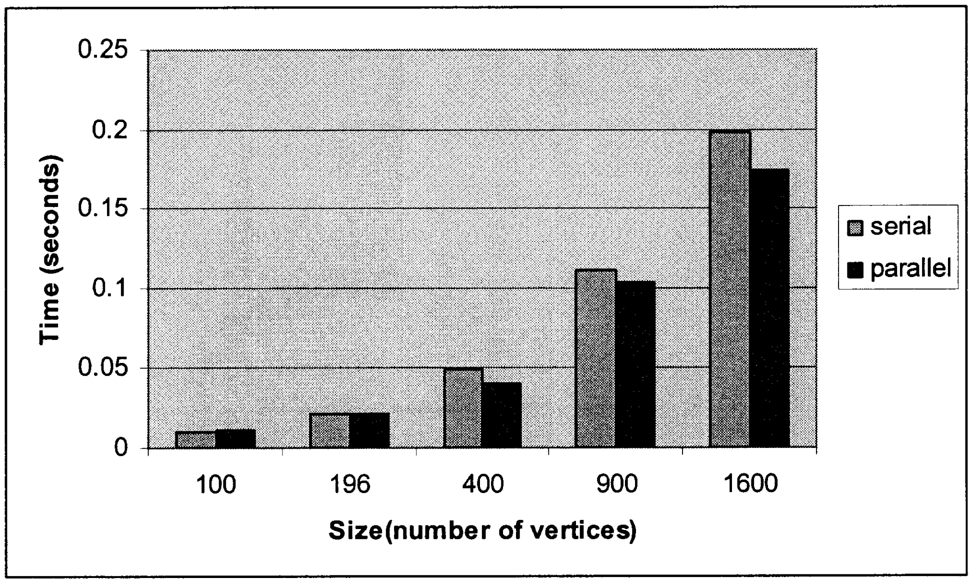


Figure 5.1 Time needed to partition various graphs.

For the smallest graph (100 vertices) the single processor is faster than the two processor system. But for the largest graph, the two-processor system it is 12% faster. Figure 5.2 shows the speedup (or slowdown) achieved.

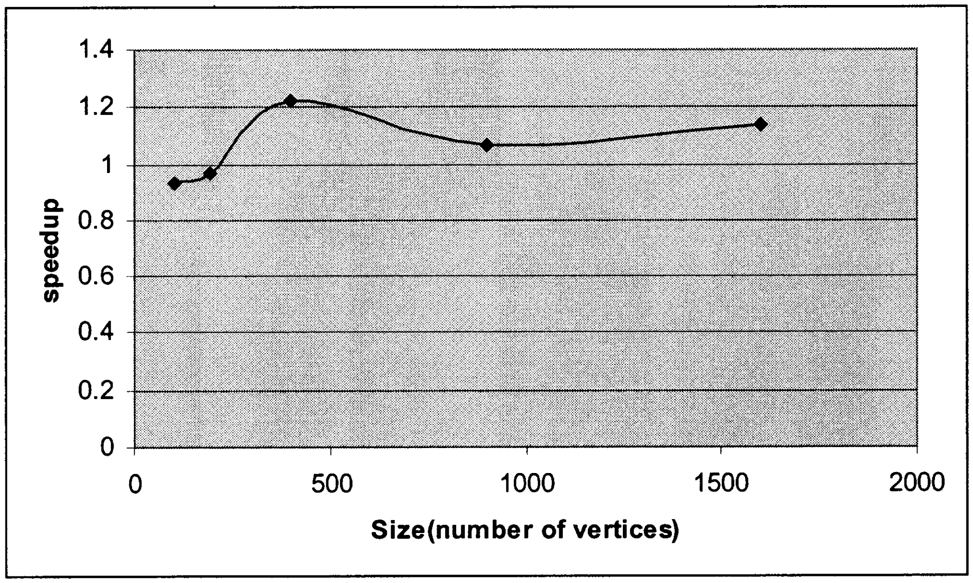


Figure 5.2 Speedup achieved from single to two-processor system.

Basically, each of the two processors executes the single processor code as well the following overhead:

- The coloring phase is a pure overhead on the parallel implementation as it is not needed in the serial implementation. So, the speedup is sensitive to the time consumed in this phase.
- Computing matching based on graph coloring which consumes extra time checking for colors.
- Interface vertices computations. Vertices on the border of a processor domain require extra computations.
- Communicating interface vertices data (color, match … etc.).
- Synchronization between the two processors. Some steps need to be synchronized and this consumes more time.

All of the above factors (except coloring) depend on the percentage of the interface vertices of the whole graph. In general, as the size of the graph increases, the percentage of the interface vertices will decrease. If the percentage of interface vertices is decreased, we are able to get more speedup from our multiprocessor system. Table 5.1 shows the relation between speedup and the percentage of interface vertices.

All the results shown are consistent except for the 20x20 graph. The reason for the high speedup is that it uses just five colors to color 90% percent of the graph. The number of colors depends on random values using Luby's algorithm [18] and [11]. According to the distribution of the random numbers among vertices, a graph could have more or less colors. If less colors were used to color most of the graph, the coloring phase and other dependent phases will need less time, and hence achieve better speedup.

Table 5.1 Relation between Interface Percentage and Speedup.

Graph	**Size**	**Number of colors**	**Interface percentage**	**Speedup**
10x10	100	6	20%	Slowdown (7%)
14x14	196	6	14%	Slowdown (3%)
20x20	400	5	10%	22%
30x30	900	6	6.7%	7%
40x40	1600	6	5%	14%

5.3 Quality of Partitions

The ultimate measure for partitions quality is the (weighted) sum of the edge-cut. This is found by summing the weights of the edges that connect vertices which belong to different partitions. Hence, a partition with lower edge-cut is considered a better partition (other constraints apply; see the section about partition balance below).

It is hard for any parallel implementation to beat the serial one in getting lower edge-cut (although possible). The reason is that the single-processor design has a global view of the whole graph which helps in getting better partitions. On the other hand, the graph is distributed between processors for the parallel multiprocessor implementation. So, in general, the goal of parallel graph partitioning is to achieve quality comparable to the quality produced by the basic serial algorithm.

Table 5.2 illustrates the edge-cut found for the single- and two-processors systems and compares it to the optimal case.

Table 5.2 edge-cut achieved for different designs.

Graph	**Number of edges**	**One processor**	**Two processor**	**Optimal**
10x10	360	14	16	10
14x14	728	20	25	14
20x20	1520	30	35	20
30x30	3480	44	56	30
40x40	6240	60	73	40

The algorithm used for refinement and improving the quality of the partition is Greedy Refinement [11]. Although this algorithm is fast, it may fail to escape local minimum of the edge-cut. In general, this algorithm as seen in the above table produce partitions of a reasonable quality and still do it fast.

5.4 Partitions Balance

Partitions balance reflects how the weights of the vertices are distributed among the partitions. If the total sum of weights of vertices is W and we have N partition, a perfectly balanced partitioning would give every partition W/N of the weights.

In general, partitions balance is a constraint rather than a goal (like low edge-cut). So, a graph partitioning system will impose a maximum and minimum limit for partitions weight. Usually, this is done as a variation percentage limit of the above W/N value.

Partition balance is harder to maintain when there is a huge variation in vertex weights, especially when moving heavy-weighted vertices between partitions.

In the experiments, the partitions produced were well balanced and most of them where within 2% of the W/N value.

5.5 Memory Usage

It is desirable to perform memory-aware graph partitioning. As well, the multilevel scheme saves all the graphs at different coursing level. Usually, the coarser graph is half or slightly more of the finer graph.

Table 5.3 shows the number of vertices the test graphs after the coarsening step.

Table 5.3 Number of Vertices at the Next Finer Level.

Graph	**Original number of vertices**	**Coarse number of vertices (serial)**	**Coarse number of vertices (parallel)**
10x10	100	50	55
14x14	196	98	110
20x20	400	200	222
30x30	900	450	490
40x40	1600	800	872

As it is apparent in table 5.3, single processor system can shrink the input graph more effectively (and save more memory). The reason is that in the parallel scheme based on coloring, not all the vertices participate in matching and are left unmatched to the next level.

The multiprocessor system needs extra memory to hold redundant information about interface processors, i.e., interface nodes information is duplicated among the boundary processors. So, the extra needed memory is proportional to the number of interface vertices.

In general, the parallel implementation requires more memory than the serial one, but the memory requirements per processor is much less in the parallel formulation.

5.6 Conclusions and Future Work

In this work, a two-processor system for parallel graph partitioning on an FPGA was designed. A single processor design was also implemented to be the compared with the two-processor system.

A speedup up to 22% from the single processor design was achieved. This speedup is strongly dependent on the percentage of interface vertices and on the number of colors used.

One can think that the optimal speedup for moving from a single processor to two processors would be 2; i.e., the dual processor system should be twice faster. That is not the case for parallel graph partitioning. The reason for that is the new step of coloring the graph before coarsening which is not needed in single processor design. This step is considered a pure overhead in the parallel formulation, especially, it accounts for 20-25% of the total partitioning time.

Comparing our two-processor design with a single processor one with "optional coloring" step would be up to 1.7 faster; which is a very attractive speedup. And for other phases, we reached a point near the limits of speedup.

The gain of moving from two processors to three or more is expected to be much better that moving for single to two-processor system. Future work will be to implement a three-processor system.

A huge part of the work was dedicated for coding and debugging, because of the use of our own parallel programming constructs. For future applications, we may look for available parallel libraries. We also plan to consider code development using Java language on an FPGA-based Java processor.

APPENDIX A

GRAPH INPUT AND OUTPUT FILE FORMATS

A.1 Introduction

Two ways were used to input a test graph into our graph partitioning software. The first method is to read it from a graph input file with a specific format. This format will be described in this appendix. This is the same format used by well-known graph partitioning software, such as METIS [15] and Chaco [16]. Here we provide an overview of this file format. More details are available in [15] and [16].

The second method is using a generated input graph. All generated graphs were of the type NxN grids. Although the "regular" graphs generated with this method have less practical value, we used them to study the relationship between the size of the graph and performance metrics in a consistent manner. The graph generation program is described in appendix B.

A.2 Input File Format

Input files contain information about the adjacency structure of the graph and the weights of vertices and edges. When either or both of the vertex and edge weights are unavailable, they get the default value of 1. Actually, most input test graphs have default weight values of 1.

We will illustrate the format with a simple example. Figure A.1 shows a sample graph with eight vertices.

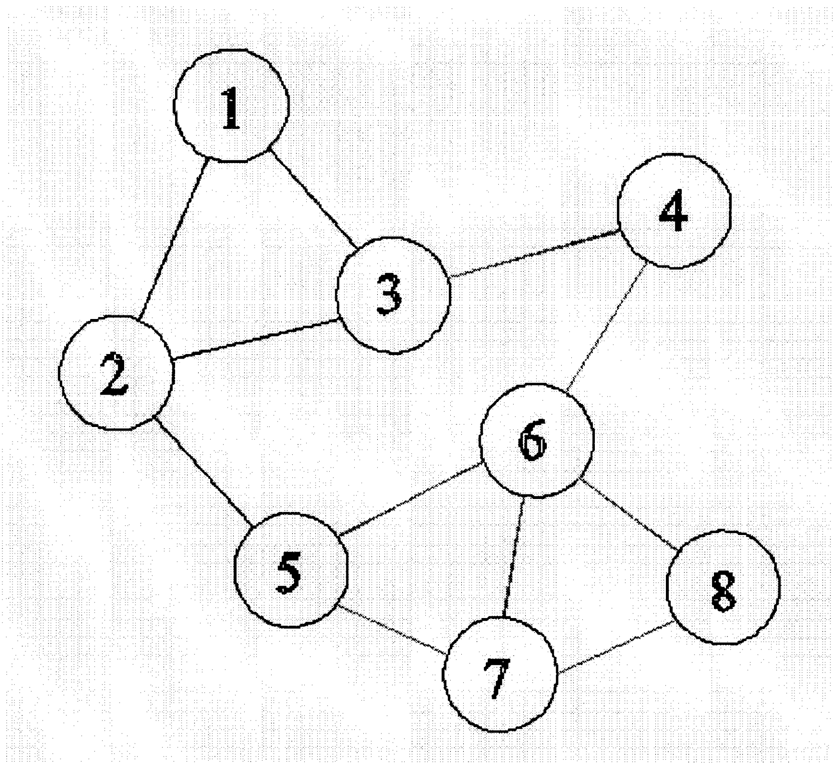


Figure A.1 Sample input graph.

The eight vertices are numbered from 1 to 8. The corresponding file that describes this graph "sample.graph" is shown in figure A.2.

```
8 11

2 3
1 3 5
1 2 4
3 6
2 6 7
4 5 7 8
5 6 8
6 7
```

Figure A.2 Content of "sample.graph"

The first line contains two numbers. The first one specifies the number of vertices and the other value is the number of edges. After that every line lists the neighbor vertices for every vertex. The first line lists the neighbors of vertex 1, the second line lists the neighbors of vertex 2, and so on. Since no weights are specified, all vertex and edge weights will have the default value of 1.

A.3 Output File Format

The output file format is simple. It just specifies for every vertex the partition number assigned by the graph partitioning program. Figure A.3 shows an example partitioning of the above sample graph.

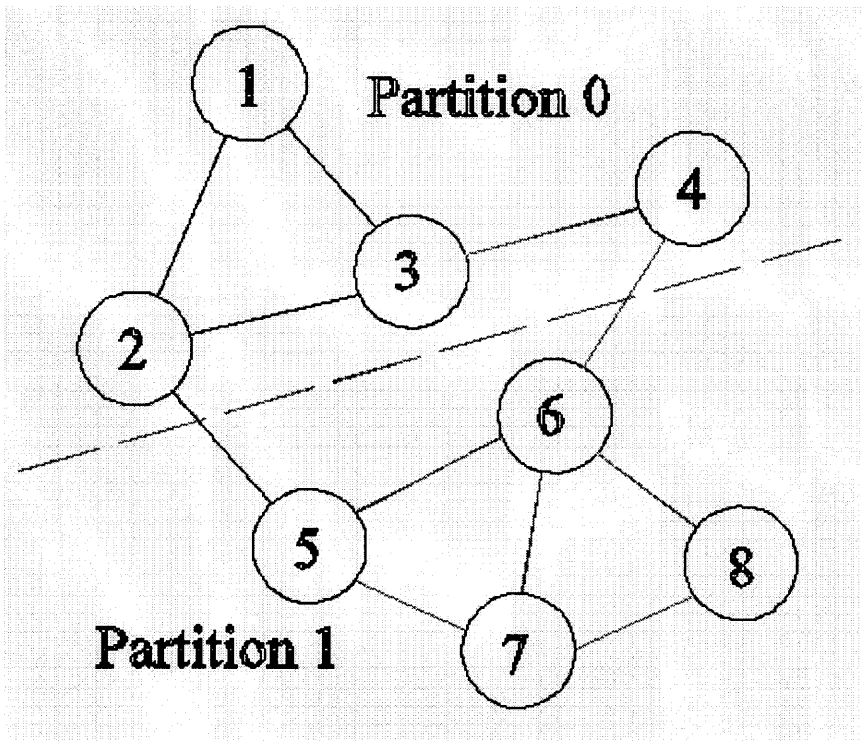


Figure A.3 Sample partitioning.

For the above partitioning, figure A.4 shows the content of the output partition file. It has a line for every vertex and that line specifies the assigned partition for it (either partition 0 or partition 1 in our case).

```
0
0
0
0
1
1
1
1
```

Figure A.4 Content of "sample.part".

APPENDIX B

GRAPH GENERATOR PROGRAM

B.1 Introduction

To study the relation between graph size and other performance metrics, a family of similar graphs (with different sizes) was used. These graphs were NxN grids. A program was written to generate directly the "xadj" and "adjncy" data structures (to be used directly as input to the graph partitioning software).

B.2 Generator Source Code Listing

The following is listing of the C++ code for the graph generator program:

```
#include <iostream.h>
#include <fstream.h>

#define N 10  // The generated graph will be NxN

int main()
{
        ofstream fout;
        fout.open("grid.dat");

        int xadj[N*N+1];
        int adjncy[4*N*(N-1)];
        xadj[0] = 0;

        int counter = 0;
        int j=0;
        for(int i=0; i<(N*N); i++)
        {
                counter = 0;
                if(i >= N)
                {
                        counter++;
                        adjncy[j] = i-N;
                        j++;
                }
                if(i%N != 0)
                {
                        counter++;
                        adjncy[j] = i-1;
                        j++;
                }
                if(i%N != (N-1))
                {
                        counter++;
                        adjncy[j] = i+1;
                        j++;
                }
                if(i < N*N-N)
                {
                        counter++;
                        adjncy[j] = i+N;
                        j++;
                }
                xadj[i+1] = xadj[i] + counter;
        }

        fout<<"xadj["<<N*N+1<<"] = {0";
        for(int k=0; k<N*N; k++)
        {
                if(k%N == 0)
                        fout<<endl;
                fout<<","<<xadj[k+1];
        }
        fout<<"}"<<endl<<endl;

        fout<<"adjncy["<<4*N*(N-1)<<"] = {";
        for(k=0; k<4*N*(N-1); k++)
        {
                if(k%N == 0)
                        fout<<endl;
                fout<<adjncy[k]<<",";
        }
        fout<<"}";

        return 0;
}
```

APPENDIX C

ALTERA DEVELOPMENT ENVIRONMENT

C.1 Introduction

Several Altera tools were used throughout the development process. Altera SOPC Builder was used for hardware design using IP cores. Altera Quartus II was used for compiling the design and for programming it on the FPGA. Also, Altera Nios IDE was used for the software development and for running it on the FPGA. In the following sections we provide more details.

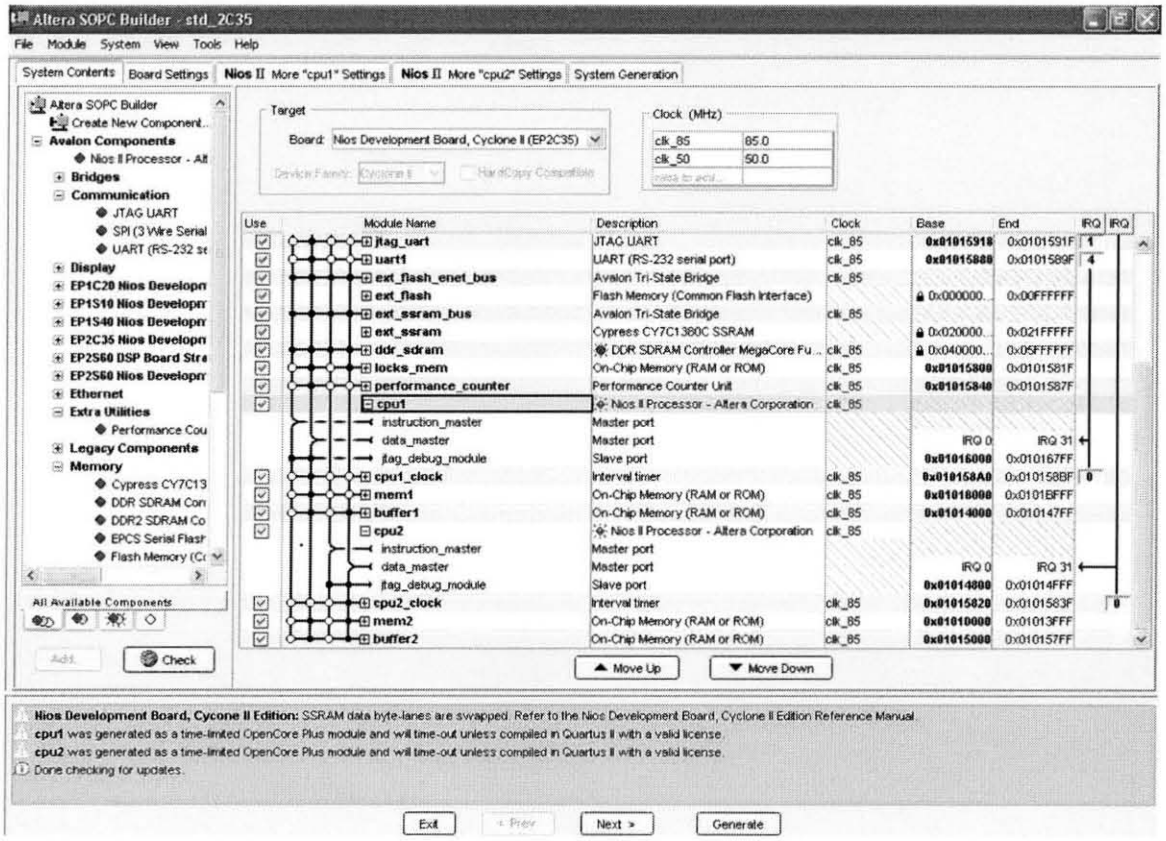
C.2 SOPC Builder

Altera SOPC Builder 5.0 was used for the hardware design. It provides a user friendly environment for building systems from the list of available (IP) components. The user can choose the hardware components to be included and add them to the design. Then the components are connected using Altera Avalon Bus Fabric. Connecting components could be done easily by clicking the mouse on the intersection of the connectivity matrix between the desired components. Figure C.1 shows SOPC Builder user interface. The connectivity matrix is apparent in the figure.

Each component has properties that can be edited and customized easily; one can change the baud rate of a UART, increase the size of a local memory component, or even change advanced properties of the processor core used.

SOPC Builder can assign addresses for devices automatically, although the designer can customize it. The designer can also assign the IRQ level number for different devices (lower IRQ values have priority), with the processor clock IRQ having the highest level of priority (value of zero).

When the design is ready an HDL code representing the system is generated by SOPC Builder. For more details, the designer can refer to the SOPC documentation available from Altera [17].



Board: Nios Development Board, Cyclone II (EP2C35)

Clock	(MHz)
clk_85	85.0
clk_50	50.0

Use	Module Name	Description	Clock	Base	End	IRQ	IRQ
✓	jtag_uart	JTAG UART	clk_85	0x01015918	0x0101591F	1	
✓	uart1	UART (RS-232 serial port)	clk_85	0x01015880	0x0101589F	4	
✓	ext_flash_enet_bus	Avalon Tri-State Bridge	clk_85				
✓	ext_flash	Flash Memory (Common Flash Interface)		0x000000...	0x00FFFFFF		
✓	ext_ssram_bus	Avalon Tri-State Bridge	clk_85				
✓	ext_ssram	Cypress CY7C1380C SSRAM		0x020000...	0x021FFFFF		
✓	ddr_sdram	DDR SDRAM Controller MegaCore Fu...	clk_85	0x040000...	0x05FFFFFF		
✓	locks_mem	On-Chip Memory (RAM or ROM)	clk_85	0x01015800	0x0101581F		
✓	performance_counter	Performance Counter Unit	clk_85	0x01015840	0x0101587F		
✓	cpu1	Nios II Processor - Altera Corporation	clk_85				
	instruction_master	Master port					
	data_master	Master port		IRQ 0	IRQ 31		
	jtag_debug_module	Slave port		0x01016000	0x010167FF		
✓	cpu1_clock	Interval timer	clk_85	0x010158A0	0x010158BF	0	
✓	mem1	On-Chip Memory (RAM or ROM)	clk_85	0x01018000	0x0101BFFF		
✓	buffer1	On-Chip Memory (RAM or ROM)	clk_85	0x01014000	0x010147FF		
✓	cpu2	Nios II Processor - Altera Corporation	clk_85				
	instruction_master	Master port					
	data_master	Master port		IRQ 0	IRQ 31		
	jtag_debug_module	Slave port		0x01014800	0x01014FFF		
✓	cpu2_clock	Interval timer	clk_85	0x01015820	0x0101583F		0
✓	mem2	On-Chip Memory (RAM or ROM)	clk_85	0x01010000	0x01013FFF		
✓	buffer2	On-Chip Memory (RAM or ROM)	clk_85	0x01015000	0x010157FF		

```
Nios Development Board, Cycone II Edition: SSRAM data byte-lanes are swapped. Refer to the Nios Development Board, Cyclone II Edition Reference Manual.
cpu1 was generated as a time-limited OpenCore Plus module and will time-out unless compiled in Quartus II with a valid license.
cpu2 was generated as a time-limited OpenCore Plus module and will time-out unless compiled in Quartus II with a valid license.
Done checking for updates.
```

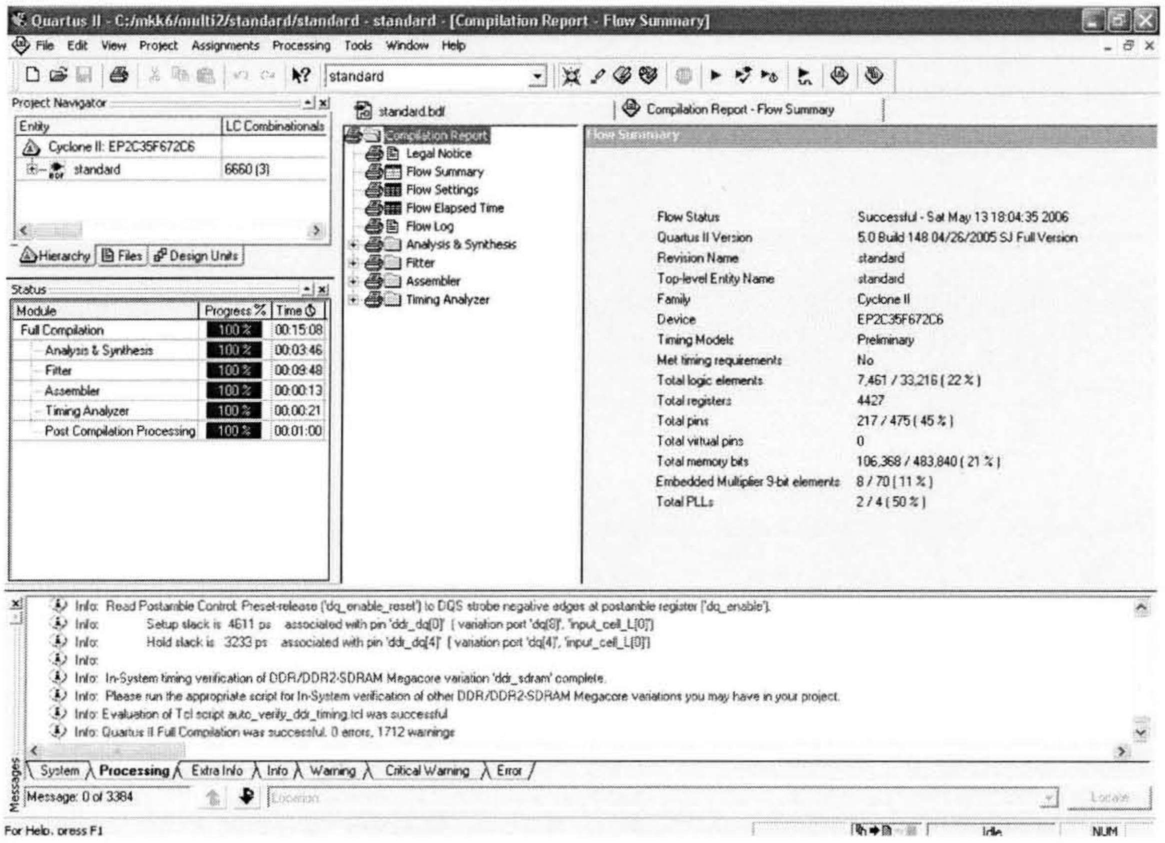
Figure C.1 Altera SOPC Builder user interface.

C.3 Quartus II

Altera Quartus II 5.0 was used for compiling, fitting, and programming the design on the FPGA.

After successfully generating the design in SOPC Builder, it should be compiled by Quartus II before being downloaded to the board. But before that, the designer should verify pin assignment and other board settings such as the name of the FPGA used. Figure C.2 shows the screen of a Quartus II compilation summary.

When the compilation is done, an "design.sof" file is generated. This ".sof" file is used by Quartus II programmer to configure the FPGA with the required design.



Entity	LC Combinationals
Cyclone II: EP2C35F672C6	
standard	6660 (3)

Status

Module	Progress %	Time
Full Compilation	100 %	00:15:08
Analysis & Synthesis	100 %	00:03:46
Fitter	100 %	00:09:48
Assembler	100 %	00:00:13
Timing Analyzer	100 %	00:00:21
Post Compilation Processing	100 %	00:01:00

Flow Status	Successful - Sat May 13 18:04:35 2006
Quartus II Version	5.0 Build 148 04/26/2005 SJ Full Version
Revision Name	standard
Top-level Entity Name	standard
Family	Cyclone II
Device	EP2C35F672C6
Timing Models	Preliminary
Met timing requirements	No
Total logic elements	7,461 / 33,216 (22 %)
Total registers	4427
Total pins	217 / 475 (45 %)
Total virtual pins	0
Total memory bits	106,368 / 483,840 (21 %)
Embedded Multiplier 9-bit elements	8 / 70 (11 %)
Total PLLs	2 / 4 (50 %)

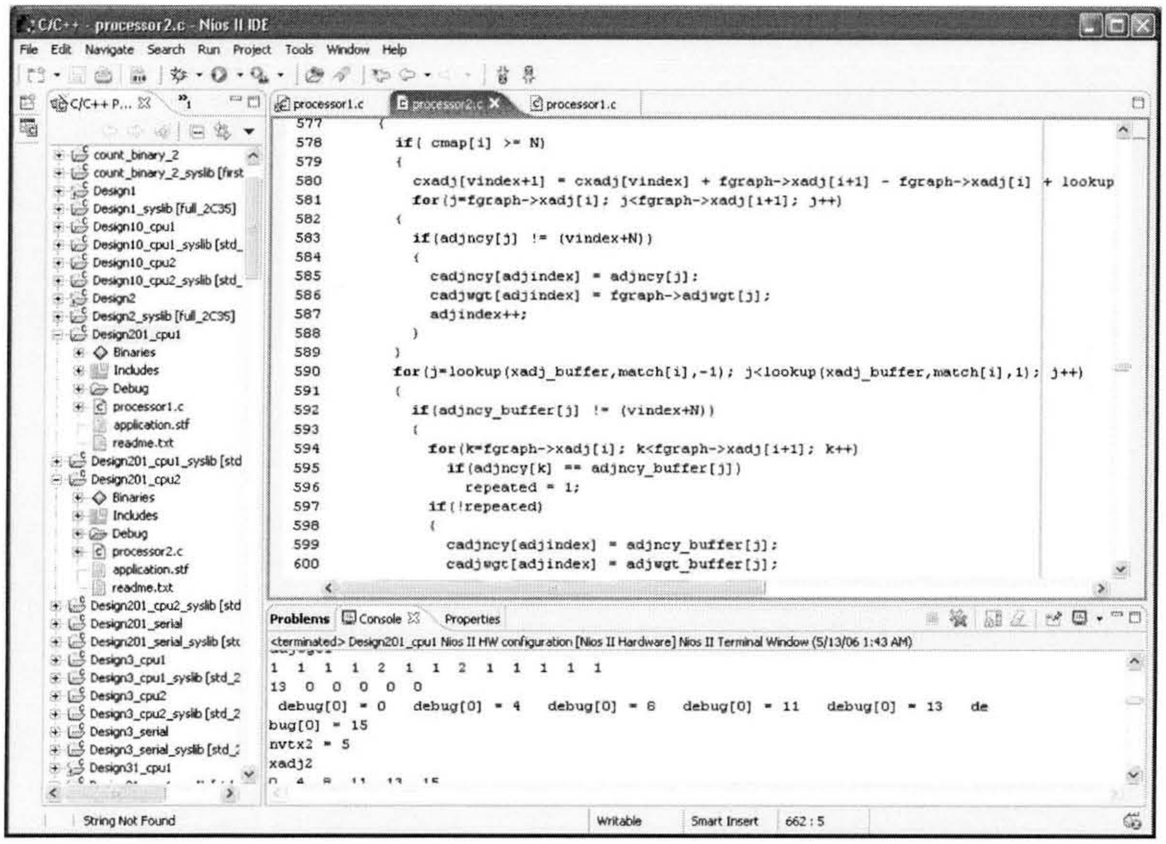
```
Info: Read Postamble Control: Preset-release ('dq_enable_reset') to DQS strobe negative edges at postamble register ('dq_enable').
Info:         Setup slack is  4611 ps    associated with pin 'ddr_dq[0]' ( variation port 'dq[8]', 'input_cell_L[0]')
Info:         Hold slack is  3233 ps    associated with pin 'ddr_dq[4]' ( variation port 'dq[4]', 'input_cell_L[0]')
Info:
Info: In-System timing verification of DDR/DDR2-SDRAM Megacore variation 'ddr_sdram' complete.
Info: Please run the appropriate script for In-System verification of other DDR/DDR2-SDRAM Megacore variations you may have in your project.
Info: Evaluation of Tcl script auto_verify_ddr_timing.tcl was successful
Info: Quartus II Full Compilation was successful. 0 errors, 1712 warnings
```

Message: 0 of 3384

Figure C.2 Quartus II sample compilation screen.

C.4 Nios II IDE

Altera Nios II IDE (Integrated Development Environment) was used for software development purposes. This IDE provides usual features found in traditional IDEs, such as compiling, building, debugging, and running the code. Figure C.3 shows the user interface of the Nios II IDE.


C/C++ - processor2.c - Nios II IDE
File Edit Navigate Search Run Project Tools Window Help
C/C++ P...
processor1.c
processor2.c
processor1.c
count_binary_2
count_binary_2_syslib [first
Design1
Design1_syslib [Full_2C35]
Design10_cpu1
Design10_cpu1_syslib [std_
Design10_cpu2
Design10_cpu2_syslib [std_
Design2
Design2_syslib [Full_2C35]
Design201_cpu1
Binaries
Includes
Debug
processor1.c
application.stf
readme.txt
Design201_cpu1_syslib [std
Design201_cpu2
Binaries
Includes
Debug
processor2.c
application.stf
readme.txt
Design201_cpu2_syslib [std
Design201_serial
Design201_serial_syslib [stc
Design3_cpu1
Design3_cpu1_syslib [std_2
Design3_cpu2
Design3_cpu2_syslib [std_2
Design3_serial
Design3_serial_syslib [std_
Design31_cpu1
Problems
Console
Properties
String Not Found
Writable
Smart Insert
662 : 5

```
        {
          if( cmap[i] >= N)
          {
            cxadj[vindex+1] = cxadj[vindex] + fgraph->xadj[i+1] - fgraph->xadj[i] + lookup
            for(j=fgraph->xadj[i]; j<fgraph->xadj[i+1]; j++)
          {
            if(adjncy[j] != (vindex+N))
            {
              cadjncy[adjindex] = adjncy[j];
              cadjwgt[adjindex] = fgraph->adjwgt[j];
              adjindex++;
            }
          }
          for(j=lookup(xadj_buffer,match[i],-1); j<lookup(xadj_buffer,match[i],1); j++)
          {
            if(adjncy_buffer[j] != (vindex+N))
            {
              for(k=fgraph->xadj[i]; k<fgraph->xadj[i+1]; k++)
                if(adjncy[k] == adjncy_buffer[j])
                  repeated = 1;
              if(!repeated)
              {
                cadjncy[adjindex] = adjncy_buffer[j];
                cadjwgt[adjindex] = adjwgt_buffer[j];
```

```
<terminated> Design201_cpu1 Nios II HW configuration [Nios II Hardware] Nios II Terminal Window (5/13/06 1:43 AM)
1  1  1  1  2  1  1  2  1  1  1  1  1
13  0  0  0  0  0
 debug[0] = 0   debug[0] = 4   debug[0] = 8   debug[0] = 11   debug[0] = 13   de
bug[0] = 15
nvtx2 = 5
xadj2
0  4  8  11  13  15
```

Figure C.3 Nios II IDE user interface.

The available compiler provides great support for the C programming language and its standard libraries, but provides limited support to the C++ programming language and its libraries. For example, it does not support C++ classes or C++ style dynamic memory allocation.

The IDE provides three main types of running the compiled C code. The first one is running it on the actual hardware. The second is running it on Nios II instruction set simulator. The third is running the code on Nios II Modelsim.

Using the IDE, the developer can specify the memory to be used for the five sections of a program (text, read-only data, read-write data, heap, and stack). Altera online documentation provides more details about using the Nios II IDE.

APPENDIX D

SOURCE CODE

The following listing is the graph partitioning code for processor 2 (processor 1 code is similar):

```
#include <stdlib.h>
#include <stdio.h>
#include "system.h"

#define N 800      // half number of vertices
#define M 3120     // half number of edges
#define L 40       // the value in LxL graph grid

typedef struct
{
  short nvtxs;
  short nedges;
  short ncolors;
  short ninterfaces;

  short* xadj;
  short* adjncy;
  short* vwgt;
  short* adjwgt;

  short* interface;
  short* color;
  short* cmap;
  short* part;

  short* ID;
  short* ED;

  short edgecut;
  short* pwgt;

  short ninter2;
  short* inter1;
  short* inter2;

  void* cgraph;
  void* fgraph;

} Graph;

void initGraph(Graph* graph, short nvtxs, short nedges)
{
  graph->nvtxs = nvtxs;
  graph->nedges = nedges;
  graph->ncolors = 0;
  graph->ninterfaces = 0;

  graph->xadj = (short*)malloc((nvtxs+1) * sizeof(short));
  graph->adjncy = (short*)malloc(nedges * sizeof(short));
  graph->vwgt = (short*)malloc(nvtxs * sizeof(short));
  graph->adjwgt = (short*)malloc(nedges * sizeof(short));

  graph->interface = (short*)malloc(nvtxs * sizeof(short));
  graph->color = (short*)malloc(nvtxs * sizeof(short));
  graph->cmap = (short*)malloc(nvtxs * sizeof(short));
  graph->part = (short*)malloc(nvtxs * sizeof(short));

  graph->ID = (short*)malloc(nvtxs * sizeof(short));
  graph->ED = (short*)malloc(nvtxs * sizeof(short));
  graph->pwgt = (short*)malloc(2 * sizeof(short));

  graph->cgraph = NULL;
  graph->fgraph = NULL;
}

void synchronize()
{
  short* lock = (short*)LOCKS_MEM_BASE;
  *lock = 1;
  while(*lock);
}

short lookup(short* buffer, short vertex, short offset)
{
  short i=0;
  while(buffer[i] != vertex)
    i = i + 2;
  return buffer[i+offset];
}

short getIndex(short* buffer, short vertex)
{
  short i=0;
  while(buffer[i] != vertex)
    i++;
  return i;
}

void pre(Graph* graph)
{
  short* mem1 = (short*)MEM1_BASE;
  short* mem2 = (short*)MEM2_BASE;

  short* inter_buffer = mem1 + 1;
  short* inter_buffer2 = mem2 + 1;

  short i, j;

  for(i=0; i<graph->nvtxs; i++)
  {
    graph->interface[i] = 0;
    for(j=graph->xadj[i]; j<graph->xadj[i+1]; j++)
```

```
      if(graph->adjncy[j] < N && !graph->interface[i])
      {
        graph->interface[i] = 1;
        graph->ninterfaces++;
      }
  }

  graph->inter1 = (short*)malloc(graph->ninterfaces * sizeof(short));

  mem2[0] = graph->ninterfaces;
  j=0;
  for(i=0; i<graph->nvtxs; i++)
    if(graph->interface[i])
    {
      graph->inter1[j] = i+N;
      inter_buffer2[j] = i+N;
      j++;
    }

  synchronize();  // always after sending

  graph->inter2 = (short*)malloc(mem1[0] * sizeof(short));
  graph->ninter2 = mem1[0];
  for(i=0; i<mem1[0]; i++)
    graph->inter2[i] = inter_buffer[i];
}

void color(Graph* graph)
{
  short* mem1 = (short*)MEM1_BASE;
  short* mem2 = (short*)MEM2_BASE;
  //short* debug = mem1 + 100;

  short i, j, k, max;
  short nColored = 0;
  short currentColor = 0;

  short* rnd = (short*)malloc(graph->nvtxs*sizeof(short));
  for(i=0; i<graph->nvtxs; i++)
    rnd[i] = (rand()+1) & 32767;


  for(i=0; i<graph->nvtxs; i++)
    graph->color[i] = -1;

  short* color_lock = (short*)LOCKS_MEM_BASE + sizeof(short);
  *color_lock = 1;
  while(*color_lock)
  {
    for(i=0; i<graph->ninterfaces; i++)
    {
      mem2[j] = rnd[graph->inter1[i]-N];
    }


    synchronize();
    for(i=0; i<graph->nvtxs; i++)
    {
    if(graph->color[i] == -1)
    {

      max = rnd[i];

      for(j=graph->xadj[i]; j<graph->xadj[i+1]; j++)
      {
        if(graph->adjncy[j] < N)
        {
          k = mem1[getIndex(graph->inter2, graph->adjncy[j])];
          if(k >= max)
          {
            max = k;
            break;
          }
        }
        else
        {
          if(rnd[graph->adjncy[j]-N] >= max)
          {
            max = rnd[graph->adjncy[j]-N];
            break;
          }
        }
      }

      if(max == rnd[i] && max != -1)
      {
        graph->color[i] = currentColor;
        nColored++;
      }
    }
    }
    currentColor++;
    for(i=0; i<graph->nvtxs; i++)
      if(graph->color[i] != -1) rnd[i] = -1;

    synchronize();
  }
  graph->ncolors = currentColor;
  free(rnd);
}

void coarsen(Graph* fgraph, Graph* cgraph)
{
  short* mem1 = (short*)MEM1_BASE;
  short* mem2 = (short*)MEM2_BASE;

  short* cmap_buffer = mem1 + 3*(fgraph->ninterfaces)*sizeof(short);
  short* xadj_buffer = mem1 + 6*(fgraph->ninterfaces)*sizeof(short);
  short* adjncy_buffer = xadj_buffer + 3*(fgraph-
>ninterfaces)*sizeof(short);
  short* adjwgt_buffer = adjncy_buffer + 5*(fgraph-
>ninterfaces)*sizeof(short);
  short* cmap_buffer2 = mem2 + 3*(fgraph->ninterfaces)*sizeof(short);
  short* xadj_buffer2 = mem2 + 6*(fgraph->ninterfaces)*sizeof(short);
  short* adjncy_buffer2 = xadj_buffer2 + 3*(fgraph-
>ninterfaces)*sizeof(short);
  short* adjwgt_buffer2 = adjncy_buffer2 + 5*(fgraph-
>ninterfaces)*sizeof(short);

  short i, j, k, myMatch;
  short cnvtxs = N;
  short* match = (short*)malloc((fgraph->nvtxs) * sizeof(short));
  short* cmap = (short*)malloc((fgraph->nvtxs) * sizeof(short));
  for(i=0; i<fgraph->nvtxs; i++)
    match[i] = -1;

  short index;
  for(k=0; k<fgraph->ncolors; k++)
  {
    for(i=0; i<fgraph->ninterfaces; i++)
      {
        mem2[i] = match[fgraph->inter1[i]-N];
      }

    synchronize();

    for(i=0; i<(fgraph->ninter2); i++)
    {
      if(mem1[i] >= N)
      {
        if(match[mem1[i]-N] == -1 || match[mem1[i]-N] == fgraph-
>inter2[i])
        {
          match[mem1[i]-N] = fgraph->inter2[i];
        }
        else
        {
          mem1[i] = fgraph->inter2[i];
        }
      }
    }

    synchronize(); // newly added -- should be added before

    for(i=0; i<fgraph->ninterfaces; i++)
    if(fgraph->inter1[i] == mem2[i])
      match[fgraph->inter1[i]-N] = mem2[i];

    for(i=0; i<fgraph->nvtxs; i++)
    {
      if(fgraph->color[i] == k)
      {
        if(match[i] == -1)
        {
          myMatch = i + N;
          for(j=fgraph->xadj[i]; j<fgraph->xadj[i+1]; j++)
            if(fgraph->adjncy[j] >= N)
            {
              if(match[fgraph->adjncy[j] - N] == -1)
              {
                myMatch = fgraph->adjncy[j];
                break;
              }
            }
            else
            {
              if(mem1[getIndex(fgraph->inter2, fgraph->adjncy[j])] == -
1)
              {
                myMatch = fgraph->adjncy[j];
                break;
              }
            }
          //cmap[i] = cmap[myMatch] = cnvtxs++;
          match[i] = myMatch;
          if(myMatch >= N) match[myMatch - N] = i + N;
        }
      }
    }

  }
  /////////////////////////////////
    for(i=0; i<fgraph->ninterfaces; i++)
      {
        mem2[i] = match[fgraph->inter1[i]-N];
      }

    synchronize();

    for(i=0; i<(fgraph->ninter2); i++)
    {
      if(mem1[i] >= N)
      {
        if(match[mem1[i]-N] == -1 || match[mem1[i]-N] == fgraph-
>inter2[i])
        {
          match[mem1[i]-N] = fgraph->inter2[i];
        }
        else
        {
          mem1[i] = fgraph->inter2[i];
        }
      }
    }

  ////////////////be careful when match = -1
  for(i=0; i<fgraph->nvtxs; i++)
    if(match[i] == -1)
      match[i] = i+N;

  synchronize();  // I added this just for testing

  /////////////////////////////////////////
  for(i=0; i<fgraph->nvtxs; i++)
    cmap[i] = -1;

  j=0;
  for(i=0; i<fgraph->nvtxs; i++)
  if(cmap[i] == -1)
  {
    if(match[i] < N)
    {
      if(match[i]%2 == 0)
      {
        cmap[i] = cnvtxs;
        cnvtxs++;
        cmap_buffer2[j] = match[i];
        j++;
        cmap_buffer2[j] = cmap[i];
        j++;
      }
    }
    else
    {
      cmap[i] = cmap[match[i] - N] = cnvtxs;
      cnvtxs++;
    }
  }
    synchronize(); // added after sending messages

  j=0;
  for(i=0; i<N; i++)
    if(match[i] < N)
    {
    if(match[i]%2 == 1)
    {
      cmap[cmap_buffer[j]-N] = cmap_buffer[j+1];
      j=j+2;
    }
    }
  j=0;
  for(i=0; i<fgraph->ninterfaces; i++)
    {
      cmap_buffer2[j] = fgraph->inter1[i];
      j++;
      cmap_buffer2[j] = cmap[fgraph->inter1[i]-N];
      j++;
    }

  synchronize();


  short* cxadj = (short*)malloc((N+1) * sizeof(short));
  short* cvwgt = (short*)malloc(N* sizeof(short));
  short* cadjncy = (short*)malloc(M * sizeof(short));
  short* cadjwgt = (short*)malloc(M * sizeof(short));
  cxadj[0] = 0;

///////////////////////////////////////////cvwgt

  for(i=0; i<fgraph->nvtxs; i++)
    cvwgt[i] = 0;

  j=0;
  for(i=0; i<fgraph->nvtxs; i++)
  {
    if(cmap[i] >= N)
      cvwgt[cmap[i]-N] += fgraph->vwgt[i];
    else
    {
      xadj_buffer2[j] = cmap[i];
      j++;
      xadj_buffer2[j] = fgraph->vwgt[i];
      j++;
    }
  }
    xadj_buffer2[j] = -1;

  synchronize(); // always after sending

  j=0;
  while(xadj_buffer[j] != -1)
  {
    cvwgt[xadj_buffer[j]-N] += xadj_buffer[j+1];
    j = j+2;
  }

  ////////////////////////////////////////cvwgt


  short* visited = (short*)malloc((fgraph->nvtxs) * sizeof(short));
  for(i=0; i<fgraph->nvtxs; i++)
    visited[i] = 0;


  short vindex = 0;
  short adjindex = 0;
  short repeated = 0;

  short* adjncy = (short*)malloc((fgraph->nedges) * sizeof(short));

  for(i=0; i<fgraph->nedges; i++)
    if(fgraph->adjncy[i] < N)
    {
      adjncy[i] = lookup(cmap_buffer, fgraph->adjncy[i], 1);
    }
    else
      adjncy[i] = cmap[fgraph->adjncy[i]-N];

    xadj_buffer[0] = -1;
    xadj_buffer[1] = 0;

  j=2;
  for(i=0; i<fgraph->ninterfaces; i++)
    {
      xadj_buffer2[j] = fgraph->inter1[i];
      j++;
      xadj_buffer2[j] = fgraph->xadj[fgraph->inter1[i]-N+1] - fgraph-
>xadj[fgraph->inter1[i]-N];
      j++;
    }
  synchronize(); // always after sending

  for(i=2; i<(fgraph->ninterfaces)+2; i++)
  {
    xadj_buffer[2*i+1] += xadj_buffer[2*i-1];
  }

  k=0;
  for(i=0; i<fgraph->ninterfaces; i++)
    {
      for(j=fgraph->xadj[fgraph->inter1[i]-N]; j<fgraph->xadj[fgraph-
>inter1[i]-N+1]; j++)
      {
        adjncy_buffer2[k] = adjncy[j];
        adjwgt_buffer2[k] = fgraph->adjwgt[j];
        k++;
      }
    }

  synchronize(); // always after sending


  for(i=0; i<fgraph->nvtxs; i++)
  {
    if(!visited[i])
    {
      visited[i] = 1;

      if(match[i] == (i+N))
      {
        cxadj[vindex+1] = cxadj[vindex] + fgraph->xadj[i+1] - fgraph-
>xadj[i];
        for(j=fgraph->xadj[i]; j<fgraph->xadj[i+1]; j++)
        {
          for(k=cxadj[vindex]; k<adjindex; k++)
            if(adjncy[j] == cadjncy[k])
            {
              cxadj[vindex+1]--;
              cadjwgt[k] += fgraph->adjwgt[j];
              repeated = 1;
              break;
            }
          if(!repeated)
          {
            cadjncy[adjindex] = adjncy[j];
            cadjwgt[adjindex] = fgraph->adjwgt[i];
            adjindex++;
          }
          repeated = 0;
        }
        vindex++;
      }
      else if(match[i] >= N)
      {
        visited[match[i]-N] = 1;
        cxadj[vindex+1] = cxadj[vindex] + fgraph->xadj[i+1] - fgraph-
>xadj[i] + fgraph->xadj[match[i]+1-N] - fgraph->xadj[match[i]-N] - 2;

        for(j=fgraph->xadj[i]; j<fgraph->xadj[i+1]; j++)
        {
          if(adjncy[j] != (vindex+N))
          {
            cadjncy[adjindex] = adjncy[j];
            cadjwgt[adjindex] = fgraph->adjwgt[j];
            adjindex++;
          }
        }
        for(j=fgraph->xadj[match[i]-N]; j<fgraph->xadj[match[i]+1-N];
j++)
        {
          if(adjncy[j] != (vindex+N))
          {
            for(k=fgraph->xadj[i]; k<fgraph->xadj[i+1]; k++)
              if(adjncy[k] == adjncy[j])
                repeated = 1;
            if(!repeated)
            {
              cadjncy[adjindex] = adjncy[j];
              cadjwgt[adjindex] = fgraph->adjwgt[j];
              adjindex++;
            }
            else
            {
              cxadj[vindex+1]--;
              k=cxadj[vindex];
              while(cadjncy[k] != adjncy[j])
                k++;
              cadjwgt[k] += fgraph->adjwgt[j];
            }
          }
          repeated = 0;
        }
                vindex++;

      }
      else
      {
        if( cmap[i] >= N)
        {
          cxadj[vindex+1] = cxadj[vindex] + fgraph->xadj[i+1] - fgraph-
>xadj[i] + lookup(xadj_buffer, match[i], 1) - lookup(xadj_buffer,
match[i], -1) - 2;
          for(j=fgraph->xadj[i]; j<fgraph->xadj[i+1]; j++)
        {
          if(adjncy[j] != (vindex+N))
          {
            cadjncy[adjindex] = adjncy[j];
            cadjwgt[adjindex] = fgraph->adjwgt[j];
            adjindex++;
          }
        }
        for(j=lookup(xadj_buffer,match[i],-1);
j<lookup(xadj_buffer,match[i],1); j++)
        {
          if(adjncy_buffer[j] != (vindex+N))
          {
            for(k=fgraph->xadj[i]; k<fgraph->xadj[i+1]; k++)
              if(adjncy[k] == adjncy_buffer[j])
                repeated = 1;
            if(!repeated)
            {
              cadjncy[adjindex] = adjncy_buffer[j];
              cadjwgt[adjindex] = adjwgt_buffer[j];
              adjindex++;
            }
            else
            {
              cxadj[vindex+1]--;
              k=cxadj[vindex];
              while(cadjncy[k] != adjncy[j])
                k++;
              cadjwgt[k] += fgraph->adjwgt[j];
            }
          }
          repeated = 0;
        }
        vindex++;

        }
      }
    }
  }
////////////////////////////////////
  initGraph(cgraph, vindex, cxadj[vindex]);
  for(i=0; i<=cnvtxs; i++)
  {
    cgraph->xadj[i] = cxadj[i];
    debug[N+1+i] = cxadj[i];
  }
  for(i=0; i<cgraph->nedges; i++)
    cgraph->adjncy[i] = cadjncy[i];
  for(i=0; i<cgraph->nedges; i++)
  {
    cgraph->adjwgt[i] = cadjwgt[i];
    //mem1[i] = cgraph->adjwgt[i];
  }

  for(i=0; i<cnvtxs; i++)
    cgraph->vwgt[i] = cvwgt[i];
  for(i=0; i<fgraph->nvtxs; i++)
    fgraph->cmap[i] = cmap[i];
////////////////////////////////////

  for(i=0; i<cgraph->nedges; i++)
  {
    mem1[i] = cadjwgt[i];
    cgraph->adjwgt[i] = mem1[i];
  }
  fgraph->cgraph = cgraph;
  cgraph->fgraph = fgraph;

    synchronize(); // for testing

  free(match);
  free(cmap);
  free(cxadj);
  free(cvwgt);
  free(adjncy);
  free(cadjncy);
  free(cadjwgt);

  synchronize(); // for test
}

void GrowBisection(Graph* graph)
{
  short* xadj2 = (short*)MEM1_BASE;
  short* adjncy2 = xadj2 + 1000;
  short* vwgt2 = adjncy2 + 5000;
  short* adjwgt2 = (short*)MEM2_BASE;
  short* nvtxs2 = adjwgt2 + 5000;
  short* edgecut2 = nvtxs2 + 1;
  short* part2 = edgecut2 + 1;
  short* pwgt2 = part2 + 1000;

  short i;

  for(i=0; i<=graph->nvtxs; i++)
  {
    xadj2[i] = graph->xadj[i];
    debug[0] = graph->xadj[4];
  }

  for(i=0; i<graph->nvtxs; i++)
    vwgt2[i] = graph->vwgt[i];

  for(i=0; i<graph->nedges; i++)
    adjncy2[i] = graph->adjncy[i];

  for(i=0; i<graph->nedges; i++)
    adjwgt2[i] = graph->adjwgt[i];

  nvtxs2[0] = graph->nvtxs;

  synchronize();

  synchronize();

  for(i=0; i<graph->nvtxs; i++)
    graph->part[i] = part2[i];

  graph->edgecut = edgecut2[0];
  graph->pwgt[0] = pwgt2[0];
  graph->pwgt[1] = pwgt2[1];
```

```
}

void uncoarsen(Graph* fgraph, Graph* cgraph)
{
  short i, j;

  fgraph->edgecut = cgraph->edgecut;
  fgraph->pwgt[0] = cgraph->pwgt[0];
  fgraph->pwgt[1] = cgraph->pwgt[1];

  short* cmap_buffer = (short*)MEM1_BASE;
  short* cmap_buffer2 = (short*)MEM2_BASE;

  j=0;
  for(i=0; i<fgraph->ninterfaces; i++)
  {
    {
      if(fgraph->cmap[fgraph->inter1[i]-N] >= N)
      {
      cmap_buffer2[j] = fgraph->cmap[fgraph->inter1[i]-N];
      j++;
      cmap_buffer2[j] = cgraph->part[fgraph->cmap[fgraph->inter1[i]-N]-
N];
      j++;
      }
    }
  }

  synchronize(); // always after sending

  for(i=0; i<fgraph->nvtxs; i++)
  {
    if(fgraph->cmap[i] >= N)
      fgraph->part[i] = cgraph->part[fgraph->cmap[i]-N];
    else
      fgraph->part[i] = lookup(cmap_buffer, fgraph->cmap[i], 1);
  }

  synchronize(); // just in case
}

void GreedyRefine(Graph* graph)
{
  short i, j, k;
  short myPart;

  short* delta_pwgt = (short*)MEM1_BASE;
  short* part_buffer = delta_pwgt + 2;

  short* delta_edgecut = (short*)MEM2_BASE;
  short* delta_pwgt2 = delta_edgecut + 1;
  short* part_buffer2 = delta_pwgt2 + 2;

  *delta_edgecut = 0;
  delta_pwgt2[0] = 0;
  delta_pwgt2[1] = 0;
  for(i=0; i<graph->ncolors; i++)
  {
    //debug[0] = i;
    for(k=0; k<graph->ninterfaces; k++)
    {
        part_buffer2[k] = graph->part[graph->inter1[k]-N];
    }

    synchronize();  // always after sending

    graph->pwgt[0] += delta_pwgt[0];
    graph->pwgt[1] += delta_pwgt[1];

    synchronize(); // just to be sure previous value is read

    delta_pwgt2[0] = 0;
    delta_pwgt2[1] = 0;


    for(j=0; j<graph->nvtxs; j++)
    {
      graph->ID[j] = 0;
      graph->ED[j] = 0;

      myPart = graph->part[j];
      for(k=graph->xadj[j]; k<graph->xadj[j+1]; k++)
      {
        if(graph->adjncy[k] >= N)
        {
          if(graph->part[graph->adjncy[k]-N] == myPart)
            graph->ID[j] += graph->adjwgt[k];
          else graph->ED[j] += graph->adjwgt[k];
        }
        else
        {
          if(part_buffer[getIndex(graph->inter2, graph->adjncy[k])] ==
myPart)
          graph->ID[j] += graph->adjwgt[k];
          else graph->ED[j] += graph->adjwgt[k];
        }
      }
    }
    for(j=0; j<graph->nvtxs; j++)
    {
      if(graph->color[j] == i)
      {
        myPart = graph->part[j];
        if(graph->ED[j] - graph->ID[j] == 0)
        {
          if(graph->pwgt[myPart] > N)
          {
            graph->part[j] = 1 - graph->part[j];
            graph->pwgt[myPart] -= graph->vwgt[j];
            graph->pwgt[1-myPart] += graph->vwgt[j];
            delta_pwgt2[myPart] -= graph->vwgt[j];
            delta_pwgt2[1-myPart] += graph->vwgt[j];
          }
        }
        else if((graph->ED[j] - graph->ID[j] > 0) && (graph-
>pwgt[myPart] - graph->vwgt[j] > N - N/4))
        {
          graph->part[j] = 1 - graph->part[j];
          *delta_edgecut -= graph->ED[j] - graph->ID[j];
          graph->pwgt[myPart] -= graph->vwgt[j];
          graph->pwgt[1-myPart] += graph->vwgt[j];
          delta_pwgt2[myPart] -= graph->vwgt[j];
          delta_pwgt2[1-myPart] += graph->vwgt[j];
        }
      }
    }
  }

  synchronize();
  graph->pwgt[0] += delta_pwgt[0];
  graph->pwgt[1] += delta_pwgt[1];
}

void post(Graph* graph)
{
  short* part_buffer = (short*)MEM2_BASE;
  int i;
  for(i=0; i<graph->nvtxs; i++)
    part_buffer[i] = graph->part[i];

  synchronize();
}

int main()
{

  Graph fgraph, cgraph;
  initGraph(&fgraph, N, M);

  short i,j;

  pre(&fgraph);

  color(&fgraph);

  coarsen(&fgraph, &cgraph);

  GrowBisection(&cgraph);

  uncoarsen(&fgraph, &cgraph);

  GreedyRefine(&fgraph);

  //post(&fgraph);

  return 0;
}
```

REFERENCES

1. Schloegel, K., Karypis, G., & Kumar, V. (2000). Graph partitioning for high performance scientific simulations. Morgan Kaufmann.

2. Kernighan, B., Lin, S. (1969). An efficient heuristic procedure for partitioning graphs. The Bell System Technical Journal.

3. Karypis, G., Kumar, V. (1998). Multilevel k-way partitioning scheme for irregular graphs. Journal of Parallel and Distributed Computing.

4. Pellerin, D., Thibault, S. (2005). Practical FPGA programming in C. Prentice Hall.

5. Compton, K., Hauck, S. (2002). Reconfigurable computing: A survey of systems and software. ACM Computing Surveys.

6. Wolf, W. (2004). FPGA-based system design. Prentice Hall.

7. Altera Corporation. (2005). Nios development board Cyclone II edition reference manual. www.altera.com

8. Karypis, G., Kumar, V. (1998). A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing.

9. Karypis, G., Kumar, V. (1998). A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. Journal of Parallel and Distributed Computing.

10. Walshaw, C., Cross, M. (1999). Parallel optimization algorithms for multilevel mesh partitioning. Technical Report.

11. Karypis, G., Kumar, V. (1998). Parallel multilevel k-way partitioning scheme for irregular graphs. SIAM Journal on Scientific Computing.

12. Karypis, G., Kumar, V. (1997). A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. SIAM Conference on Parallel Processing for Scientific Computing.

13. Altera Corporation. (2005). Nios II processor reference handbook. www.altera.com.

14. Altera Corporation. (2005). Quartus II handbook. Volume 5: Altera embedded peripherals. www.altera.com

15. Karypis, G., Kumar, V. (1998). METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. Technical Report.

16. Hendrickson, B., Leland, R. (1995). The Chaco User's Guide version 2.0. Technical Report.

17. Alter Corporation. (2005). Quartus II 5.0 software handbook, volume 4: SOPC Builder. Technical Report.

18. Karpinski, M., Rytter, W. (1998). Fast parallel algorithms for graph matching problems. Oxford Science Publications.