PRAM and the Ear Decomposition Algorithm

TJHSST Senior Research Project

2007-2008

Alex Valentin

The History of PRAM

PRAM refers to an abstract machine for designing algorithms in parallel. It allows for an infinite number of processors that can each access the same memory in uniform time. The first known implementation of PRAM is the University of Maryland A. James Clark School of Engineering's ParaLeap prototype 64-core supercomputer.

Steps of Ear Decomposition

Input: *A bridgeless, undirected graph G*

Output: *An ordered set of paths representing an ear decomposition of G*

begin

1. Find a spanning tree T of G

2. Root T at an arbitrary vertex r, and compute level(v) and p(v), for each vertex v ≠ r, where level(v) and p(v) are the level and the parent of v, respectively.

3. For each nontree edge, e= (u, v), compute lca(e)= lca(u,v) and level(e) = level(lca(e)). Set label(e): (level(e), s(e)), where s(e) is the serial number of e.

4. For each tree edge g, computer label(g).

5. For each nontree edge e, set P_e = {e} U {g | label(g) = label(e) }. Sort the P_e's by label(e).

end

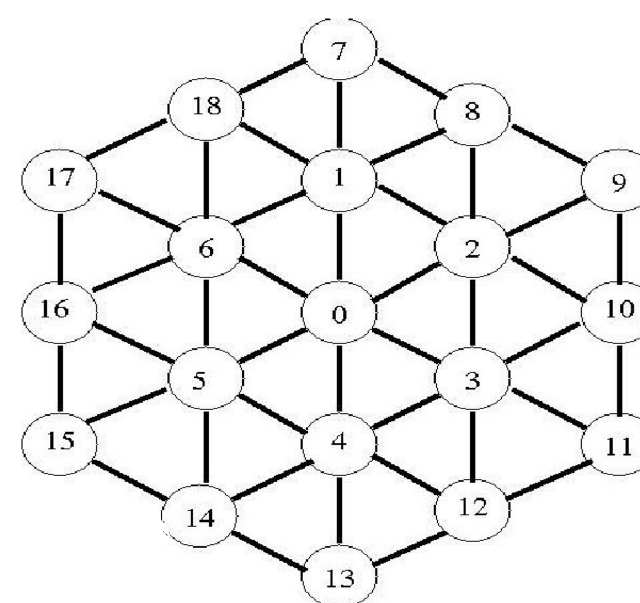
Abstract

This project takes the ear decomposition algorithm for partitioning maps and compares runtime efficiencies of different implementations. Four implementations are considered. Two implementations are run in a Parallel Random Access Machine (PRAM), while the other two are run serially. For each of these modes, one implementation uses only arrays and the other uses structures.

Procedures

The four implementations were written in XMT-C, even though the serial versions do not use the parallel capabilities. The Ear Decomposition was broken into three files, a main, span, and link. A convert file was also created for converting the input data from arrays to structures, if applicable. The span file finds a spanning tree of the data. The link file labels each edge and node with the correct ear. The main file runs the span and link files and then prints the ear of each edge and node.

Input data was given in the form of three arrays, vertices, degrees, and a two dimensional edges array. This data was quickly made with the help of the memMapCreate32 program in the XMT environment. Below is a visual of the hexagonal data set. Each data set was run on each of the four implementations of ear decomposition four times. The clock cycle counts were averaged and compared. The data is shown in Table 1 (which will arrive shortly.)

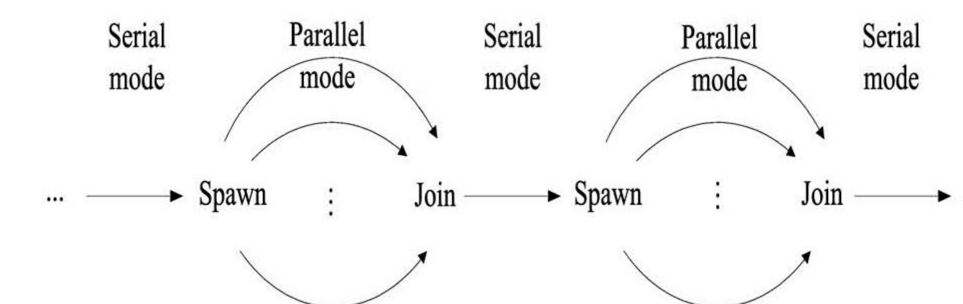


Results

The four ear decomposition implementations to be tested are the parallel using structures, parallel using only arrays, the serial using structures, and the serial using only arrays. The two parallel implementations should run faster than the serial implementations for obvious reasons. Of the two parallel implementations, the one using structures is expected to run slower because of the overhead caused by using structures. Therefore, from fastest to slowest, the implementations are parallel arrays, parallel structures, serial arrays, and serial structures.

How XMT-C Works

The language used on this supercomputer is called XMT-C, eXplicit Multi-Threaded C. Simply, the language is C with two extra methods, SPAWN and PS. The spawn method allows the programmer to use multiple processors. While any number of processors can be called for, the computer only has 64 processors to run at any given moment. The ps method, short for prefix sum, allows for the different threads to communicate with each other. The diagram below shows how XMT-C code alternates between serial mode and parallel mode.



Below is a simple XMT-C program that takes array A of length N and compacts its data into array B. N, Array A, and array B are declared in the header data.h.

```
#include "data.h"
#include <xmtc.h>
psBaseReg base;
int main()
{
base = 0;
spawn(0, N-1)
{
int step = 1;
    if( A[$] != 0)
    {
    ps(step, base);
    B[step] = A[$];
    }
}//end spawn
}//end main
```

The spawn method takes two integer arguments, which identify the range of the thread ids, inclusive. The id is referenced with the dollar sign, $. The ps method also takes two arguments, a local integer and a global psBaseReg. The local integer (*step* in the example above) is set to the global psBaseReg (*base* in the example above) and the psBaseReg is incremented by the size of step. Essentially, the ps method acts as a global counter without having to worry about concurrent writes.