XMT-C and the Ear Decompositon Algorithm
TJHSST Computer Systems Research Lab
2007-2008
Luis Alejandro 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 Decompositon

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, compute 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 a serial implementation and a parallel implementation, both of which are written in XMT-C. The number of nodes on the tested datasets span from 10 to 100. The number of edges in each span from the minimum possible to teh maximum possible. The point at which the speed up of the parallel implementation equals the overhead lag is estimated.

Procedures

Both the parallel and the serial Ear Decompositon algorithms were written in XMT-C and run on the ParaLeap. The spanning tree was found using a Breadth First Search and was rooted at the node whose value was zero. The twleve datasets tested had N nodes, where N = 10, 25, 50, and 100. For each N value, a dataset existed with E edges, where E = 25% and 75% percent of the maximum number of edges, a completely connected dataset, and N, the fewest number of edges while maintaining biconnectivity. Input data was given in the form of four arrays: vertices, antiparallel, degrees, and a two dimensional edges array. Because random data is not garanteed to be bridgeless, a special program, makeData, was written to create the datasets. First, the program connects all node in a large ring, ensuring biconnectivity. Then, the program randomly adds edges until the specified number are included. Lastly, the numbers from makeData were run through memMapCreate32 in order to use in XMT environment.

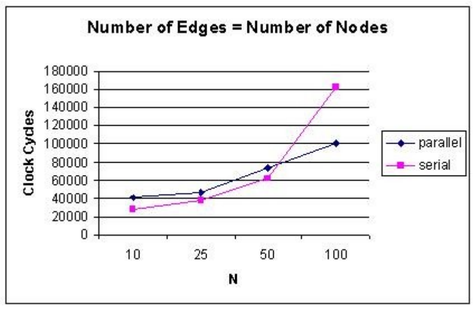
Each dataset was run three times and the clockcycles were averaged. The dataset with 50 nodes and 75 percent edges caused an unknown error, but can safely be assumed to run faster than its serial counterpart.

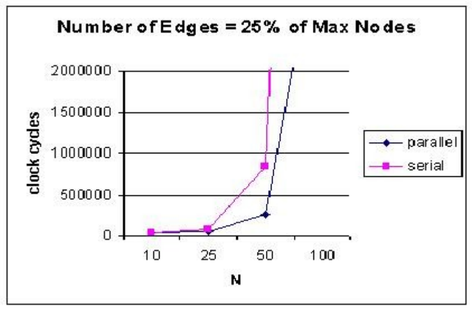
Results

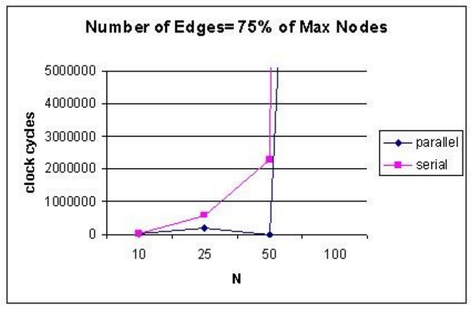
The three graphs below show how the parallel implementation of the Ear Decompostion Algorithm becomes more efficient sooner as the percentage of edges increases. When the number of edges equals the number of nodes (0%), the parallel version becomes more efficient with more than 50 nodes. When the number of edges equals 25% or higher, the number of nodes needed for the parallel to out-perform the serial code is 10.

A more efficient Depth First Search can be used when finding a spanning tree to improve both parallel and serial runtimes. A special property of ParaLeap disproportionately puts serial programs to a disadvantage. When declaring an array, all values of the array begin as random numbers. Parallel algorithms take constant time to set these values to zero, whereas serial algorithms take linear time.

Further research should test values of N significantly greater than 64, the number of processors on ParaLeap, to view the full extent of the parallel speed-up. More research can be done to determine whether an increase in ears has an effect on runtime of the Ear Decompositon Algorithm.

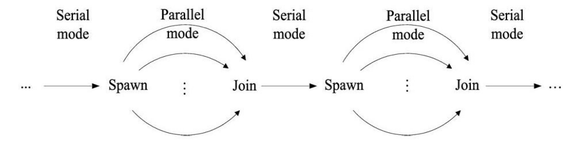






How XMT-C Works

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.