Multi-View 3D Geometry Reconstruction: Exploiting Massive Parallelism

Chaman Singh Verma
Department of Computer Sciences
University of Wisconsin, Madison

December 24, 2009

Abstract

3D geometric reconstruction from digital images captured from consumer cameras is an inexpensive, but computationally demanding application. In this experimental study, we have explored parallelism in the best known public domain software (Bundler and PMVS2) and found that massive parallelism exists at various levels that can be exploited on various computer architectures (such as multi-cores, distributed memory and GPU machines). Although in this report, we present results for multi-core and distributed memory machines, but we believe that similar results could be obtained on vector machines and GPU architectures as well. We have tested our MPI codes on a large clusters of 500 Linux nodes. Having shown almost linear scalability, we stress upon the need of improvements in sequential algorithms to enhance the quality of generated models.

1 Introduction

3D geometric reconstruction from images from digital camera is very attractive technique because digital consumer cameras are now a days quite inexpensive, extremely portable and have reasonably high resolution. Commercial large laser scanners, although provide extremely high quality triangulated models, are limited in usage because of their high cost, bulkiness and non-portability.

Another driving force for developing this technology is *Internet Imagery.* Internet is perhaps the biggest repository of images where millions of photographs of well known buildings, statues and structures can be downloaded. The major characteristic of Internet images is the diversity in every sense which provides both opportunities and challenges in the development of Internet imagery a killer future application. It is possible to find a large number of images of every conceivable viewing direction and environment conditions(sunny, cloudy, night etc). Diversity in image resolutions, exposure setting and image quality provides additional information that can be exploited to reconstruct high quality 3D models.

One of the major obstacle in reconstructing 3D models from images is high computational cost per image in the set. Depending on the image sizes and number of images, it could take many hours or days to produce acceptable quality despite using the best known algorithms on a single processor machines. In this project, we have explored parallelism (both fine grained and coarse grained) in the existing, well known public domain software that can be exploited on various computer hardware.

2 Exploiting Parallelism

In many applications, parallelism exists at many levels. Fine grain parallelism occurs at loop levels or instructions level. Such parallelism are hard to exploit and many modern compilers can automatically detect and parallelize the code. On the other hand, coarse grain or task parallelism are application dependent, simpler to implement and reason about. In many applications, coarse grained parallelism provide unlimited scalability. Fortunately, in this application both fine grains and coarse grain parallelism exists that can be exploited on modern computer hardware very efficiently.

3 Literature Survey

There is large collection of papers on the 3D reconstruction and stereo matching. But this project is influenced by ***Building Rome in a Day***[3]. In addition, we refer ***Modeling the World from Internet Photo Collection*** [4] for algorithms and techniques used in the reconstruction process.

Module	**Fine Grain Parallelism**	**Coarse Grain Parallelism**
Feature Detection	pixel level operations are independent	every image is independent so each image run on different processor
Feature Matching	Parallel KD Tree	Each image is matched against all others
SFM	Parallel Linear Algebra Parallel RANSAC	
Dense Point Cloud	Each Patch is independent and each patch is run by independent thread	Each patch is independent each patch run on different processor
Surface Reconstruction	Domain Decomposition	Domain Decomposition
Mesh Processing	Domain Decomposition	Domain Decomposition

Table 1: Parallelism in different modules of 3D reconstruction

4 Public Domain Software

Instead of developing software from scratch, we have used best known public domain software which have been provided for research purpose.

- **SIFT** is a feature detector (Lowe 2004) which provides good invariance to image transformation.
- **Bundler** Bundler takes a set of images, image features, and image matches as input, and produces a 3D reconstruction of camera and (sparse) scene geometry as output. The system reconstructs the scene incrementally, a few images at a time, using a modified version of the Sparse Bundle Adjustment package of Lourakis and Argyros as the underlying optimization engine.
- **PMVS** is a multi-view stereo software that takes a set of images and camera parameters, then reconstructs 3D structure of an object or a scene visible in the images. Only rigid structure is reconstructed, in other words, the software automatically ignores non-rigid objects such as pedestrians in front of a building. The software outputs a set of oriented points instead of a polygonal (or a mesh) model, where both the 3D coordinate and the surface normal are estimated at each oriented point.

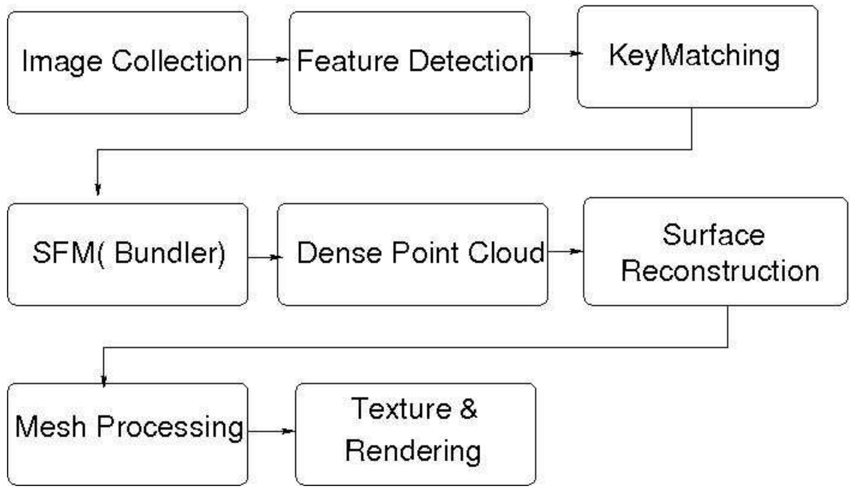

Figure 1: Pipeline of 3D model reconstruction

Data Set	**Images**	**Image Size**
Middlebury: Temple	359	640x480
Middlebury: Dino	363	640x480
Ponce: GreenDragon	24	3104x2072
Ponce: Armor	48	3504x2336
UW: BascomHall	150	1645x970
Madison: Capitol	246	1649x970

Table 2: Dataset used in the experiments

5 Results

We have implemented two most time consuming operations i.e. Feature Detection and KeyMatching using Message Passing Interface (MPI) and tested the codes on Intel Eight cores machines and a cluster of 64 nodes at Engineering Physics department at the University of Wisconsin,Madison. PMVS is already available as multithreaded code so we did not modify it. Two of the dataset are from Middlebury benchmark and two are from Ponce Research group. These four dataset have been used to compare the results with other group. We have taken large number of images of Bascom Hall and Capitol building at Madison and these two dataset are primarily used to study parallelization issues.

DataSet	**Min Features**	**Max Features**	**Mean Features**	**Total Features**
Temple	445	1144	869	314426
Dino	66	248	156	58654
Dragon	6928	9546	8194	198050
Armor	11919	55417	27764	1408775
BascomHall	3673	69907	15120	2317129
Capitol	144	28597	13700	3103375

Table 3: SIFT Features in the dataset

DataSet↓ NumThreads →	**1**	**2**	**4**	**6**
Temple	12/1.0	6.0/2.0	3.0/4.0	2.20/5.45
Dino	538/1.0	271/1.98	138/3.89	95/5.66
Dragon	831/1.0	432/1.92	237/3.50	173/4.80
BascomHall	4492/1.0	2270/1.97	1163/3.86	776/5.78
Capitol	7757/1.0	3902/1.98	1967/3.93	1366/5.67

Table 4: Features Detection on Intel Eight-Core Machine

6 Conclusions

From the results of the experiments we can conclude that

- There is almost linear scaling on the two most time consuming operations in the 3D reconstruction pipeline i.e. Feature detection and Feature matching on both Multi-Core and distributed memory machines using MPI. The parallelization of these two module is trivial task with MPI.

- Presently PMVS2 is multi-threaded and there is good scope to improve both the algorithm and the implementation on distributed memory machines.

- The point cloud generated after the PMVS2 has many outliers and has sampling errors which make it difficult for the surface re constructors to generate watertight triangulated mesh.

7 Future Work

We firmly believe that 3D reconstruction with images has great potential for future applications. But in the entire pipeline of reconstructions,it seems the

DataSet↓ NumThreads →	**1**	**4**	**7**
Temple	113m 10s	29m	16m 41s
Dino	5m 25s	4m 15s	3m 38s
Dragon	5m 25s	1m 39s	1m 7s
Armor	105m 54s	26m	16m 5s
BascomHall	6h 46m	1h 45m	62m 31s
Capitol	14h 58m	3h 50m	137m 3s

Table 5: Features Matching on Intel Eight-Core Machine

DataSet→ NumProcs ↓	**BascomHall**	**Capitol**
1	18514s	34688s
4	9257s	17258s
8	3675s	8682s
16	2520s	4300s
32	1249s	2334s
64	841s	1416s

Table 6: Features Matching on 64 Node Intel Cluster

quality of an acceptable model is primarily dependent on (1) Better feature detection (2) Dense point cloud generation. At present, the dense point cloud produced by PMVS2 has lots of scope for improvement. In many cases, it still generates large number of holes which are difficult for surface reconstructors.

In order to generate photo realistic models, it is important to apply texture mapping on the model. In future we would like to investigate applying image texture on the reconstructed models.

One of the limitations of the present algorithm is that they can not reconstruct 3D models if the model's surfaces are shinny and reflective. Probably using High Dynamic Range imaging, we can avoid input images to have shinny spots. In future, We would to explore use of HDR imaging in 3D model reconstruction.

Figure 2: Madison Capitol Building Generated with 246 images

8 MPI Source Code: KeyMatching with Load Balancing

```
#include <assert.h>
#include <time.h>
#include <string.h>
#include <sstream>
#include <fstream>
#include "keys2a.h"
#include <deque>
#include <boost/lexical_cast.hpp>

using namespace std;

#ifdef PARALLEL
#include <mpi.h>
#endif

int main(int argc, char **argv)
{
    char *list_in  = "./list.txt";
    char *file_out = "matches.init.txt";
    double ratio;

    ratio = 0.6;
    int myid = 0, numprocs = 1;
```

Figure 3: 3D Reconstruction for Ponce Armor Dataset

Figure 4: 3D Reconstruction for Ponce GreenDragon Dataset

```
    int start_proc_id = 0;

#ifdef PARALLEL
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    double  tstart = MPI_Wtime();
#endif

    unsigned char **keys;
    int *num_keys;

    /* Read the list of files */
    std::vector<std::string> key_files;

    FILE *f = fopen(list_in, "r");
    if (f == NULL) {
        printf("Error opening file %s for reading\n", list_in);
        return 1;
```

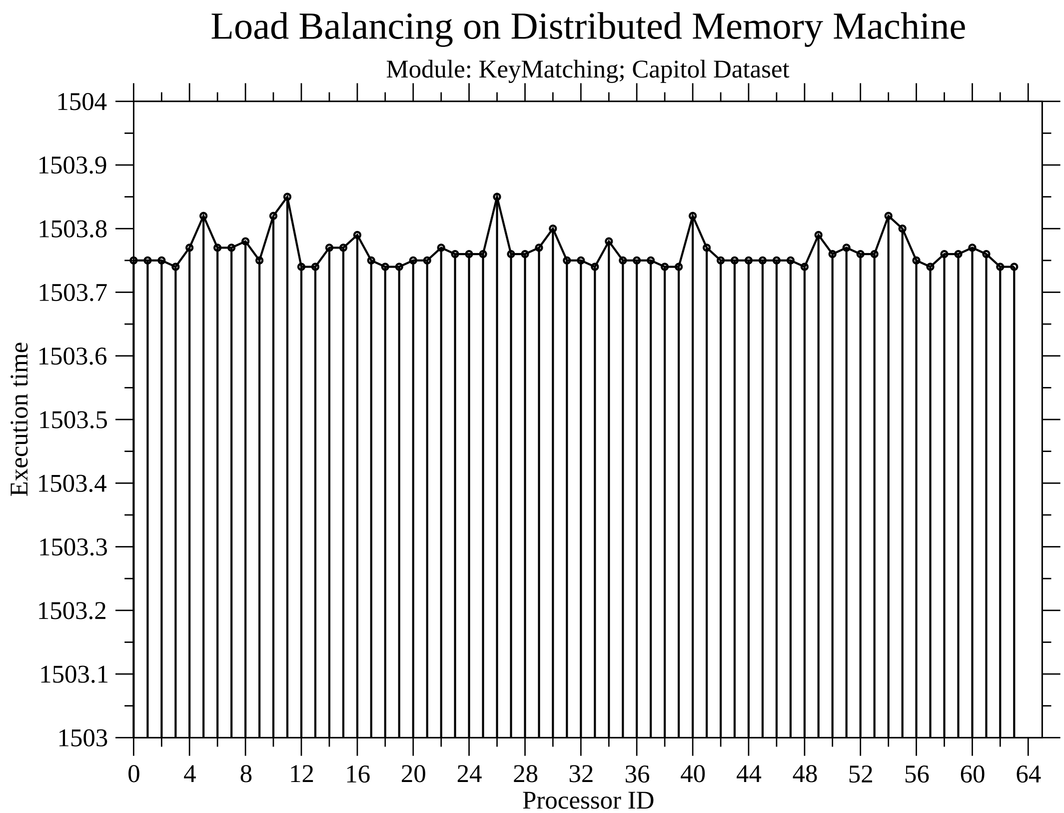

Figure 5: Dynamic load balancing on 64 nodes distributed memory machine

```
    }

    char buf[512];
    while (fgets(buf, 512, f)) {
        /* Remove trailing newline */
        if (buf[strlen(buf) - 1] == '\n')
    buf[strlen(buf) - 1] = 0;
        string  fname = std::string(buf);
size_t pos   = fname.find(".jpg");
if( pos != string::npos) {
    fname = fname.substr(0, pos);
    fname = fname + ".key";
            key_files.push_back( fname );
        }
    }
    fclose(f);

    int num_images = (int) key_files.size();
```

```
    keys = new unsigned char *[num_images];
    num_keys = new int[num_images];

    /* Read all keys */
    for (int i = 0; i < num_images; i++) {
        keys[i] = NULL;
        num_keys[i] = ReadKeyFile(key_files[i].c_str(), keys+i);
    }

    string ofilename =  file_out + boost::lexical_cast<string>(myid);
    ofstream ofile(ofilename.c_str(), ios::out);
    assert( !ofile.fail() );

    int numPieces = num_images/numprocs;
    int numindex = 0;

    int imgid, work_requester, num_images_processed = 0;
    MPI_Status mpi_status;
    if( myid == 0)
    {
        deque<int>  imgQ;
for( int i = 0; i < num_images; i++)
             imgQ.push_back( num_images-i-1 );

        for( int i = 1; i < numprocs; i++) {
      imgid = imgQ.front(); imgQ.pop_front();
      MPI_Send( &imgid, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }

        while( !imgQ.empty() ) {
      MPI_Recv( &work_requester, 1, MPI_INT, MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD,  &mpi_status);
      imgid = imgQ.front(); imgQ.pop_front();
      MPI_Send( &imgid, 1, MPI_INT, work_requester, 0, MPI_COMM_WORLD);
        }

        for( int i = 1; i < numprocs; i++)
      MPI_Recv( &work_requester, 1, MPI_INT, MPI_ANY_SOURCE, 0,
                MPI_COMM_WORLD,  &mpi_status);

        int stop_signal = -1;
        for( int i = 1; i < numprocs; i++) {
             MPI_Send( &stop_signal, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
        }
    } else {
      while(1 ) {
          MPI_Recv( &imgid, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,  &mpi_status);

         if( imgid == -1) break;
```

```
        int i = imgid;
        if (num_keys[i] )
        {
           num_images_processed++;

           /* Create a tree from the keys */
           ANNkd_tree *tree = CreateSearchTree(num_keys[i], keys[i]);

           for (int j = 0; j < i; j++) {
              if (num_keys[j] == 0) continue;

           /* Compute likely matches between two sets of keypoints */
           std::vector<KeypointMatch> matches =
              MatchKeys(num_keys[j], keys[j], tree, ratio);

           int num_matches = (int) matches.size();

           if (num_matches >= 16) {
              /* Write the pair */
              ofile << j << "  " << i << endl;

              /* Write the number of matches */
              ofile << matches.size() << endl;

              for (int i = 0; i < num_matches; i++)
  ofile << matches[i].m_idx1 << "  " << matches[i].m_idx2 << endl;
           }
  }
          delete tree;
      }
      MPI_Send( &myid, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  }
  }

  /* Free keypoints */
  for (int i = 0; i < num_images; i++) {
      if (keys[i] != NULL)
          delete [] keys[i];
  }

  delete [] keys;
  delete [] num_keys;

  ofile.close();

#ifdef PARALLEL
  double tend = MPI_Wtime();
  double elapsetime = tend-tstart;
```

```
    cout << myid << " Elapsed Time " << elapsetime << endl;

    if( myid == 0)
    {
        vector<double>  proctime(numprocs);
        vector<int>     imagecounter(numprocs);

        proctime[0] = elapsetime;
        imagecounter[0] = 0;
        for( int i = 1; i < numprocs; i++) {
             MPI_Recv( &num_images_processed, 1, MPI_INT, i, 0, MPI_COMM_WORLD,  &mpi_status);
             imagecounter[i] = num_images_processed;

             MPI_Recv( &elapsetime, 1, MPI_DOUBLE, i, 1, MPI_COMM_WORLD,  &mpi_status);
             proctime[i] = elapsetime;
        }

        string file2 =  "proctime" + boost::lexical_cast<string>(numprocs) + ".dat";
        ofstream ofile2(file2.c_str(), ios::out);
        cout << " ProcID  "  <<  "# of Images Processed " << endl;
for( int i = 0; i < numprocs; i++)  {
     ofile2 << i << "  " << proctime[i] << endl;
     cout   << i << "  " << imagecounter[i] << endl;
        }
    } else {
        MPI_Send( &num_images_processed, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
        MPI_Send( &elapsetime, 1, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD);
    }

    MPI_Barrier( MPI_COMM_WORLD );

    tend = MPI_Wtime();

    MPI_Finalize();

    if( myid == 0)
    {
      cout << "Total Execution time for KeyMatching : " << (tend - tstart) << endl;
    }
#endif

    return 0;
}
```

References

[1] Noah Snavely, Steven M. Seitz, Richard Szeliski. Photo tourism: Exploring photo collections in 3D ACM Transactions on Graphics (SIGGRAPH Proceedings), 25(3), 2006, 835-846. Multi-View Stereo for Community Photo Collections

[2] Michael Goesele, Noah Snavely, Brian Curless, Hugues Hoppe, Steven M. Seitz Multi-View Stereo for Community Photo Collections Proceedings of ICCV 2007, Rio de Janeiro, Brasil, October 14-20, 2007

[3] Building Rome in a Day Sameer Agarwal, Noah Snavely, Ian Simon, Steven M. Seitz and Richard Szeliski International Conference on Computer Vision, 2009, Kyoto, Japan.

[4] Modeling the world from Internet photo collections Noah Snavely, Steven M. Seitz, and Richard Szeliski Microsoft Research November 2008