1 Introduction

Sphere mosaicing is an extremely CPU intensive process. To mosaic a node with full size images will take several CPU days. Using current SMP technologies, we hope to spread the work across multiple processors, hence reduce the overall wall-clock time. In this report, we describe how we parallelized the code using the pthread package and present some timing results. We assume the reader is already familiar with the details of the sphere mosaicing process.

Section 2 outlines the general approach. Section 3 details the implementation. Section 4 shows some timing results. Section 5 discusses the numerical instability problem.

2 Approach

The mosaic process consists of computing convergence values for all adjacent image pairs. Our parallelization computes several image pairs at once on different processors. Since image pairs have no interdependencies within a single iteration, the parallelization preserves the correctness of the general algorithm. To facilitate distribution of image pairs, we use the producer-consumer model. One producer generates all adjacency pairs and stores it in a shared memory queue. Consumers, one on each processor, obtains an image pair from this queue and processes it. Figure 1 illustrates the setup.

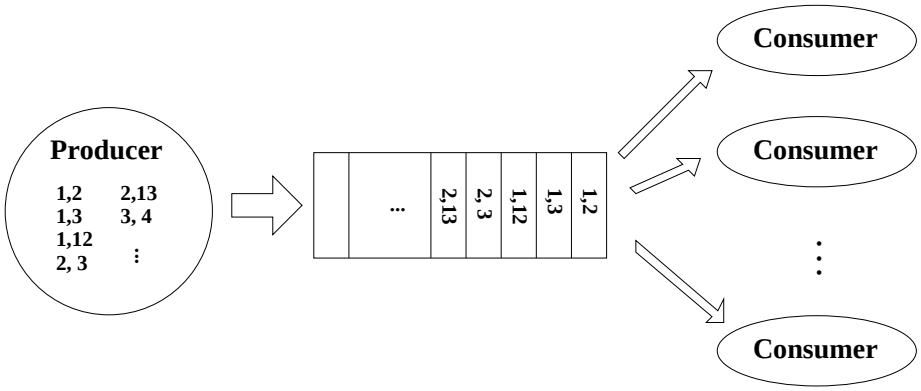

Figure 1 Producer-Consumer Model

The key computations in processing an image pair are updating the derivative vectors and the Hessian matrix. The parallelization must take precautions against two processors modifying the same entry in the derivative or the Hessian. The simplest solution is locking the data structures before using it. This lock guarantees exclusive write access to any processor; however, if the number of processors involved is large, locking the whole data structure might impose some performance penalties.

Fortunately, processing one image pair only effect very few entries, and we can divide up the derivative vector and Hessian matrix into smaller chunks and enforce mutually exclusive accesses on those chunks independently. Dividing up the Hessian does not solve the overhead associated with locking and unlocking entries for each pixel in the images. In the parallelization, we mask the overhead by accumulating all the changes locally and do one update per each image pair. This optimization allows us to run on multiple processors without significant overhead.

We did not attempt to parallelize other sections of code because computing the convergence values takes over 95% of total CPU time. While mosaicing a node with full size images, this percentage is well over 99%.

3 Implementation

We added the support for parallelization with minimum alteration to the existing code using the pthread package. Pthread allows us to run multiple consumer threads on SMP machines. In our modification, the

main processing loops are replaced by thread setup and management routines. Figure 2 shows the control flow of these thread related activities.

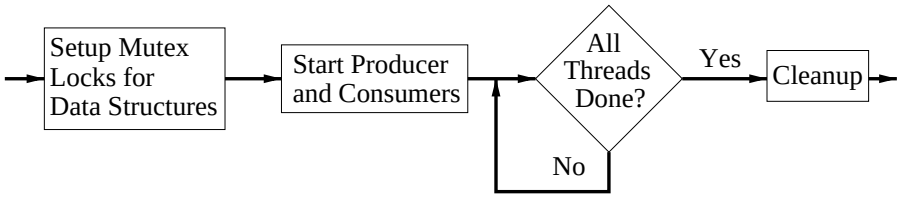

Figure 2 Control Flow for Thread Management

The producer thread is responsible for adding all adjacent image pairs into a shared memory FIFO queue. Figure 3 shows the control flow. Comparing with the old code, the producer thread is essentially the main loop where we replace the actual processing with inserting the job into the FIFO queue. To generalize the implementation, we built a template for the producer-consumer model and allow end users to specify their own actions. These customizable actions are marked *User Defined* in the template control flow.

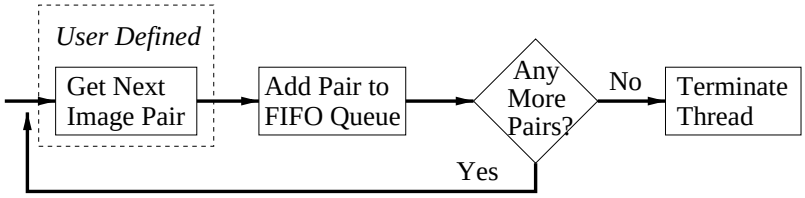

Figure 3 Control Flow for Producer Thread

The consumer thread contains the main processing routines. These routines, in the parallel version, are slightly modified so that they accumulate updates locally and only write to shared memory at the end. Figure 4 shows the control flow.

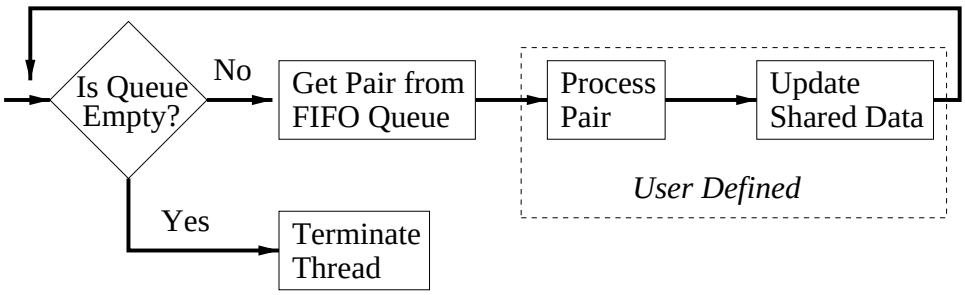

Figure 4 Control Flow for Consumer Thread

The modifications mentioned above are very well isolated. The current mosaic code allows user to switch between parallel version and sequential version without any conflicts. The appendix provides more details on how we used pthread to build these threads.

4 Results

We gathered our timing results on a SGI Onyx 2 workstation (panorama.lcs.mit.edu) with up to 4 processors and on dual processor Pentium running Linux 2.2.2 (tessarae.lcs.mit.edu). On panorama.lcs, we tested quarter size images for five different nodes using various number of processors. Table 1 summarizes the

measurements on panorama. For each entry, we give the wall clock time in seconds and the number of iterations in the gradient descent. The test nodes are from the tsq/100nodes set.

	1 Proc		2 Proc			3 Proc			4 Proc		
	time	iter	time	iter	spdup	time	iter	spdup	time	iter	spdup
Node 10	1809	14	923	14	1.96	968	20	1.87	490	14	3.69
Node 11	1176	8	545	7	2.16	369	7	3.19	302	7	3.89
Node 12	2000	14	1023	14	1.96	897	18	2.23	543	14	3.68
Node 13	1487	13	793	13	1.88	544	13	2.73	414	13	3.59
Node 14	2064	19	1091	19	1.89	786	20	2.63	569	19	3.63

Table 1 Quarter Size Images on panorama.lcs

On tessarae.lcs, we tested half size images for the five nodes on one and two processors. Table 2 summarizes the measurements on tessarae. Due to heavy usage demand for panorama.lcs, we did not get a chance to test half size images on more than two processors.

	1 Proc		2 Proc		
	time	iter	time	iter	spdup
Node 10	14527	34	7195	34	2.02
Node 11	7231	21	4783	23	1.51
Node 12	10601	24	5000	23	2.12
Node 13	7059	18	3577	18	1.97
Node 14	10194	26	5200	26	1.96

Table 2 Half Size Images on tessarae.lcs

From the above measurements, we can make the following observations,

1. The number of iterations required varies when using different number of processors.
2. If there is no change in the number of iterations, the speedup is linearly proportional to the number of processors.

Observation 2 shows our parallelization is very efficient, and we are near the optimal speedup factor. However, observation 1 is extremely disturbing because our gains in parallelization could be significantly lower due to the increase in the number of iterations needed. The next section will explore the causes of this increase in iterations.

5 Numerical Stability

The main cause of the increase in the number of iterations is numerical rounding errors. In our mosaic process, we are constantly accumulating floating point numbers. Since C language and physical hardware only have finite precision, each addition results in a small rounding error. These errors, after millions of operations, can be a significant noise in our computation.

The sequential version also accumulates floating point numbers in the same order; therefore, no variations

in the floating point error was visible. When we parallelized the code to run on multiple processors, the order of accumulating the sum is nondeterministic, and we began to observe variations in the error. Our experiments shows that different orders can result in 5% deviation from the sequential version.

The deviations contribute to differences in camera parameter calibrations and rotational angles used in the gradient descent method. We believe these differences may cause extra steps (iterations) in the descent process to find the global maximum. Of course, the differences may also lead to less steps. At current time, we are computing camera refinements to determine whether there are differences in the quality of the final spherical mosaic between the sequential version and the parallel version. Since the gradient descent is a heuristic process, we do not have an absolute optimal to compare either mosaic, sequential or parallel.

The overall issue at the heart of the problem is how do we deal with these floating point errors. We are currently considering two potential options,

1. Ignore the error and hope the final mosaic is sufficiently close to the optimal.
2. Control the error by using some form of fixed precision.

Our current code uses option 1, and the resulting mosaic appears to be satisfactory. The drawback is the nondeterminism described above. Option 2 implies that we do our computation with a certain fixed precision limit. For example, we can do floating point operations at the precision of 10^{-6}, but after each operation we only take the answer with the precision of up to 10^{-4}. This allows us to eliminate the accumulation of floating point errors. The drawback is the extra overhead in converting answers to fixed precision. Further study is needed to determine the trade off between the two options and explore other alternatives.

6 Conclusion

Our approach of processing multiple image pairs can "linearly" reduce the overall wall clock time in the spherical mosaicing process. Given that the numerical instability does not result in extra iterations, our parallelization achieves near optimal speedup with a given number of processors. There are small degradations in the speedup factor as we use more processors due to memory bandwidth and granularity of dividing up the image pairs. We do not have a clear characterization of this degradation because we do not have a computer with sufficiently large number of processors for testing.

The issue of extra iterations in the gradient descent in the parallel version is not a new problem. The primary cause is numerical instability from the floating point rounding errors. The classic scientific computing solution suggests we control the accumulation of errors by using less precision than the hardware, but trade off between the extra processing and the quality of the mosaic may not justify this solution.

Appendix

In developing the parallel version of the spherical mosaicing code, we created several "generalized" and "user-friendly" C++ classes that hides the details of the pthread package. Appendix 1 describes the thread class and its subclasses in the producer-consumer model. Appendix 2 discusses synchronized data objects and mutex locks.

Appendix 1 Thread

The base class thread hides thread creation and entry sequence from the users. It also provides a mechanism for blocking threads until other threads are done. The class definition is as follows,

```
class Thread {
public:
    int id; /* for user identification purposes */
    Tstatus status; /* status of the current thread */
public:
    Thread(); /* constructor */
    Thread(int id); /* constructor */
    void start(void); /* boiler plate for starting threads */
    static void waitfor(Thread *thrs); /* joining threads */
    static void waitfor(Thread **thrarray, int num); /* joining threads */
    virtual void run (void); /* user definable thread code */
    int isfinished(void); /* has thread terminated */
    static void setSystemScope(void); /* threads compete at system level*/
    static void setProcessScope(void); /* threads compete within user process */
};
```

The waitfor routines are used to block the calling thread until the specified thread(s) have completed. The two scope related routines control the scheduling of the threads relative to other processes. System scope means each thread is scheduled like a normal process. Process scope means all threads are scheduled as one process, thus have lower priority. To use setSystemScope, the user must have additional privileges under the SGIs.

The function run is the user defined entry point. By overriding run, users can customized their thread behavior. Create a Thread class instance does not create a running thread. The user must explicit call start to create the thread and begin execution. Thread termination and cleanup are automatically performed after the user function run returns.

The current base class does not support suspending threads. Future additions might include this feature.

A1.1 Producer Thread

The Producer subclass takes in an additional FIFO queue as a constructor parameter. This queue will hold all the data to be distributed to consumers. The class definition is

```
class Producer : public Thread {
public:
    Producer(SyncFIFOQueue *q); /* constructor */
    Producer(int id, SyncFIFOQueue *q) : Thread(id); /* constructor */
    virtual int produce(QueueElem *datum); /* adding datum to the queue */
};
```

The function produce handles insertion of a data item into the FIFO queue. The users must override the default run function to generate the data items, but they can call produce to handle the details of adding an item.

We also provide another producer thread type IterProducer that has a predefined run function. In this subclass, the user provides an iterator function generate that will return one data item each time it is called.

The definition is

```
class IterProducer : public Producer {
public:
    IterProducer(SyncFIFOQueue *q) : Producer(q)
    IterProducer(int id, SyncFIFOQueue *q) : Producer(id, q)
    virtual void run(void); /* predefined behavior */
    virtual QueueElem *generate(void); /* iterator */
};
```

In IterProducer, the thread will terminate if generate returns NULL as the return value. Therefore, the user also has control over the termination of the producer in this case.

A1.2 Consumer Thread

Similar to the producer threads, the Consumer subclass automates most of the maintenance code. The user controls the behavior of a consumer by providing a customized process routine. The class definition is

```
class Consumer : public Thread {
public:
    Consumer(SyncFIFOQueue *q);
    Consumer(int id, SyncFIFOQueue *q) : Thread(id);
    virtual void run(void); /* predefined behavior */
    virtual int process (QueueElem *); /* user defined action */
};
```

The default run function gets an item from the FIFO queue and calls the user process function. The user routine must return 0 if there is no error. In the event of a user error, the consumer thread will terminate. Furthermore, if the FIFO queue is "close" (no more data will be available in the future), the thread will also terminate.

Appendix 2 Synchronized Objects

The base class SyncObj abstracts away pthread mutex syntax and conditional variables. Each synchronized object will have one mutex and one conditional variable. The mutex is used to ensure exclusive access, if the user desires. The conditional variable is used to selectively block calling threads. One useful instance of a conditional variable is blocking caller of the FIFO queue until some new data is entered into the queue. The following is the class definition

```
class SyncObj {
public:
    SyncObj(); /* constructors */
     SyncObj();
    virtual void lock(void); /* locks the mutex */
    virtual void unlock(void); /* unlocks the mutex */
    virtual void wait(void); /* blocks the calling thread until awaken */
    virtual void wake(void); /* wakes up one thread that's blocked */
    virtual void wakeall(void); /* wakes up all blocked threads */
};
```

Although we simplified the locking and unlocking process, the user, in defining new synchronized objects, must explicit call lock and unlock. Our abstraction provides no mechanism for detecting and avoiding deadlocks. The end users must manage these issues themselves.

A2.1 Synchronized Accumulator

Accumulator is one of the most commonly used data structure. We provide a template class for this purpose.

```
template <class TYPE>
class SyncAccumulator : public SyncObj {
public:
    SyncAccumulator(TYPE initv); /* constructor */
    virtual void set(TYPE i); /* sets the value of the accumulator */
    virtual void add (TYPE i); /* adds a new value to the accumulator */
    virtual TYPE getValue(void); /* retrieves the value of the accumulator */
};
```

A user can instantiate the template with a specific data type; however, this data type must support the binary operation +.

A2.2 Synchronized FIFO Queue

For our producer-consumer model, the shared memory FIFO queue is the communication link. Because the FIFO queue is also overloaded with controlling thread termination in our model, we added some flags in the FIFO queue to indicate whether the queue is closed or not. The class definition is

```
class SyncFIFOQueue : public SyncObj {
public:
    SyncFIFOQueue();
    virtual QueueElem *get(void); /* gets the head of the queue, blocking */
    virtual int put(QueueElem *); /* puts something at the tail of the queue */
    virtual void sigdone (void); /* signal the queue is closed */
    virtual int isdone (void); /* returns whether the queue is closed */
};
```

Note, get function is blocking. The calling thread will be suspended by the system until new data arrives or the queue is closed. If a queue is already closed, get will return immediately. A peek functionality might be useful in the future, but we have not added it to the current class.