BMP-RAP: Branching Multithreaded Pipeline for Real-time Applications with Pooled Resources

Michael P. Scherer
University of Central Florida
3100 Technology Parkway
Orlando, FL 32826
michael.scherer@knights.ucf.edu

ABSTRACT

It is desirable to confirm the detection of objects or patterns in a computer vision problem using multiple hand-tailored algorithms that work differently to get a result. This is as opposed to a system such as boosting which uses a large number of weak classifiers that are generated by the machine [2]. The goal is to create a system which shall distribute work to m algorithms running on n threads. Each algorithm may take a different amount of time to execute. Because this will be for real-time applications, the system shall also guarantee a result within τ milliseconds, regardless of the number of algorithms which have completed, and give a confidence value which reflects this result. The end result will also be combined in meaningful way.

Categories and Subject Descriptors

B.2.1 [**Arithmetic and Logic Structures**]: Design Styles—*parallel, pipeline*; D.1 [**Programming Techniques**]: Concurrent Programming

General Terms

Algorithms, Design

Keywords

concurrent tree, pipeline

1. INTRODUCTION

Concurrency can be very useful for real-time systems, however it can often be difficult to manage. When dealing with robotic platforms, one may want to run data through multiple algorithms in order to confirm a result. These algorithms may need to run simultaniously in order to take advantage of the parallelism of multiple processes. In addition, there should be some mechanism to get the best result available after some amount of time has passed.

BMP-RAP is based off of the pipe-and-filter paradigm, but also includes important features such as thread and resource pooling, along with timeouts on finding the completed result. After time τ has elapsed, the system gurantees to give the best result that it has at the time.

There are many problems, particularly in computer vision, which exibit such structure naturally. For instance, most algorithms which employ a pyramid for analysis of an image in multiple scales have a structure which can be modeled in this way. In such a program, an image is fed into the system. It is then scaled successively by a factor, or alternatively some operation is performed on it several times with an increasing factor, and then the image is further operated on at each scale. At the end of the procedure, the results generated from processing each image is often collapsed down to a single set of results [4, 5].

1.1 Related Work

The pipe-and-filter model was used as a general basis for the architecture of BMP-RAP. This is because the model has very good properties when it comes to modularity, and because each filter can be considered as a single automic operation [7]. There have been many groups which have worked on refining pipe-and-filter systems to make them optimal for various applications [6].

There has also been a lot of work in pipelining a single algorithm used for data analysis in order to take advantage of parallelism [6]. Unfortunately, these systems are limited because they do not take into account pooled resources which can be shared, including threads, and also are not designed for time-critical applications.

François proposed a hybrid generic architecture for combining the benefits of a pipe and filter model with shared resources [1]. However, this system was designed such that information would continuously flow into the pipe and be output at the end. This is in contrast to the system being proposed where data flows in once per time interval, and must be processed by a given deadline.

2. PROBLEM STATEMENT

Consider a program consisting of a tree of tasks $t_0, t_1, ..., t_m$ where each task t_i passes its result to all of its children. Each task can be considered as a single automic operation on the data passed into it. All data passed into or out of a task is immutable.

Every leaf node must either have the same type or be poly-

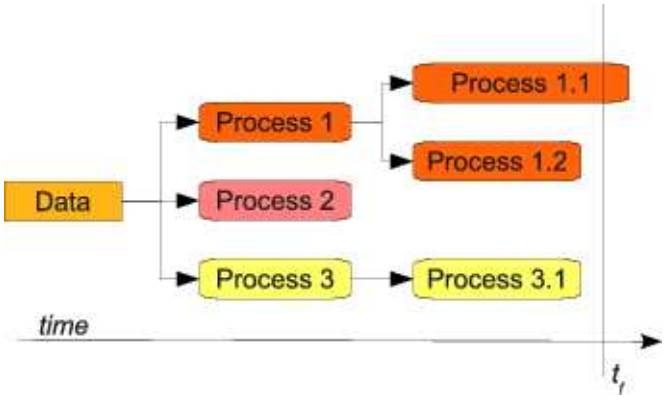

Figure 1: The process flow of a typical system

morphic to a single data type. At the end of an execution cycle, the data returned from each leaf node $\delta_0, \delta_1, ..., \delta_k$ can be combined using some algorithm into a final result δ.

Each task can be executed on any thread $T_i \in T_0, T_1, ..., T_n$. Regardless of completion status, the algorithm must be guranteed to provide a result within τ milliseconds. For non-time-critical applications, τ can be set to ∞

2.1 Problem Analysis

Clearly, this system can become bottlenecked at any place where threads need to wait for a single node to complete execution before its children can be accessed. In the special case that the system is used on an entirely serial set of branches, the system becomes useless.

3. SYSTEM OVERVIEW

In essence, the system is performing a tree traversal, where at each visited node, some operation is completed on it. For this system, there is a thread pool in order to mitigate the overhead of creating and destroying threads during runtime. A queue of nodes is created to list which branches are ready to be executed. The system is organized into three main classes: $Tree$, $Branch$, and $Thread$. An instance of $Tree$ serves as the root node, and also stores references to the thread pool. Branches are the nodes of the tree, and are inherited from to create actual functionality.

For reasons of simplicity and rigor, all data passed from a node to its child is considered to be immutable. Data is passed to a branch's children by returning it from said branch's work function. The thread will then set each child's input to the output of the branch. Finally, the thread adds each of the children to the node queue, and it starts operating on the next branch in the queue.

3.1 Using the System

To use the system, the user first creates an instance of Tree. When initialized, the user specifies the number of threads to be available in the pool, along with a function pointer to a callback function, and then begins adding child nodes to the tree. To add children, the user simply creates instances of branches of different kinds to be pieced together for the tree. On each branch, the $AddChild()$ function is called to add children to it. Children should only be added to Branches or the Tree before execution time. To activate, the user calls $SetJob$ on the instance of the Tree.

3.2 Implementing a Branch

In order to implement a Branch, one would simply inherit from the Branch class. The user must then overload a single function, and has the option to overload one additional method.

$$virtualvoid * ExecuteJob() = 0; \quad (1)$$

$$virtualvoidInterrupt() \quad (2)$$

$ExecuteJob()$ gets called when a thread has taken control of the branch. This function must be overloaded, and will do the actual work in the system. After the branch completes its work, it should return the data it wishes to pass to its children. The user may also overload the $Interrupt$ function if it wants to perform special clean-up behavior when its work function gets interrupted because of a timeout.

4. THE NODE QUEUE

In order to increase parallelism of the system, it is desirable to use a queue for adding nodes which is Wait-Free. For the purposes of this paper, a wait-free is defined algorithm as being able to finish in a bound number of steps [3]. This is an ideal requirement, because it is desirable for all threads to be working on each node as much as possible, rather than waiting to queue more nodes onto the list.

4.1 Wait-Free Node Queue Implementation

The queue is implemented as a linked-list in order to make implementation simpler. An array is defined which has one slot for each thread. When a thread wishes to enqueue an item, a reference to said item is placed in its designated location. Before attempting to enqueue its own item, it first attempts to help and enqueue its neighbor by executing a compare and swap on the tail of the list with its neighbor. After its neighbor has been committed or if its neighbor has no information to commit, then the add will attempt to commit its own item to the end of the queue.

When dequeing, a thread simply copies the head and then compare and swaps the head of the list with its next element. If the CAS passes, then it has succesfully removed the head and can continue. Otherwise, the dequeue operation will return NULL. The operation will also return NULL if the queue is empty. As a desperate consumer, a thread would spin on the dequeue operation until there is work available, yielding after a fixed number of attempts in order to allow other threads to do work and potentially enqueue more nodes.

5. RESULTS

The system was run on an Intel Quad-Core i7 with Hyperthreading, which had 8 logical cores. The machine also had 8 GB of RAM. The tested algorithms were run using BMP-RAP and also run in serial, and their run times were compared.

Actual data to be included...

6. CONCLUSIONS

To be included...

7. FUTURE WORK

To be included...

8. ACKNOWLEDGMENTS

I would like to thank the hours and efforts that Dr. Damien Dechev has put into making this class possible for all of us.

9. REFERENCES

[1] A. R. J. François. A hybrid architectural style for distributed parallel processing of generic data streams. *26th International Conference on Software Engineering (ICSEâĂŹ04)*, pages 367–376, May 2004.

[2] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *J. Comput. Syst. Sci.*, 55(1):119–139, August 1997.

[3] M. Herlihy and N. Shavit. The art of multiprocessor programming. 2008.

[4] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004.

[5] P. Perona and J. Malik. Scale-space and edge detection using anisotropic diffusion. *IEEE Trans. Pattern Anal. Mach. Intell.*, 12(7):629–639, July 1990.

[6] J. Philipps and B. Rumpe. Refinement of pipe-and-filter architectures. *FM '99: Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems*, 1:96–115, 1999.

[7] M. Shaw and D. Garlan. Software architecture - perspectives on an emerging discipline.