A Self-Optimising Coprocessor Model for Portable Parallel Image Processing

D Crookes¹, TJBrown¹, YDong¹, G McAleese², PJ Morrow², DK Roantree², ITA Spence¹

¹Department of Computer Science, Queen's University Belfast, Belfast BT7 1NN, UK
²Department of Computing Science, University of Ulster, Coleraine BT52 1SA, UK

Abstract This paper describes an approach taken to achieve both portability and efficiency in the context of image processing. It outlines the EPIC architecture (Extensible Parallel Image Coprocessor), an image processing-specific software architecture capable of efficient implementation on a range of parallel machines. EPIC is currently being implemented using a network of C40 processors.

1 Introduction

Much research in parallel programming aims to reconcile two conflicting objectives: software portability, and efficiency of parallel implementation. Often this is addressed by choosing a standard programming language (e.g. a variant of Fortran), and seeking to improve efficiency using auto-parallelisation. In fact, in addition to portability and efficiency, there is a third objective: expressive power of the programming notation. This suggests an alternative, application-specific approach. Instead of starting with full expressive power and portability (e.g. with Fortran), the EPIC project starts with portability and efficiency, and seeks to increase the expressive power of the notation EPIC provides only programming abstractions which can be implemented efficiently and, within this constraint, aims to make the abstractions more generally applicable.

This paper firstly outlines briefly the programming model which EPIC provides to an application developer. Then, the key strategy for retaining efficiency through the extensibility and self-optimising facility of the EPIC environment is presented. Finally, some outline indicators of the performance of the EPIC optimiser are given.

2 The EPIC Programming Model

EPIC aims to provide the image processing developer with the power of parallelism without the responsibility for expressing it. The basic EPIC model comprises an image coprocessor which implements the high level abstractions (in parallel), controlled by a (sequential) general-purpose host. In our implementation, the parallel coprocessor runs on a network of C40 processors. The user program runs on the host, and is written in C++. Access to the instructions of the coprocessor is provided via C++ classes which constitute the programmer's interface to the coprocessor.

The high level programming abstractions which EPIC provides for image processing are based on those of Image Algebra [1]. The power of the notation comes
primarily from the facilities for template-based neighbourhood operations. The EPIC abstractions are a significant advance on previous notations such as IAL [2] and I-BOL [3], in that EPIC supports variant (rather than static) templates, and other template constructors. The shape and size of templates can vary with image position and other parameters (though they are not image data dependent). In the context of implementation on a (distributed memory) parallel architecture, the definition of such templates provides predictability of data access. All data locations to be accessed by a neighbourhood operator can be pre-calculated (and the data pre-fetched, if appropriate) as though at the start of a BSP-style superstep [4].

Using these abstractions, it is possible to build a single template operator to carry out an image transform (such as Hadamard or Fourier) in one 'instruction' [5].

3 Efficiency through Extensibility

Although the abstract machine model makes for simpler, more portable programming, it is traditionally thought to result in inevitable inefficiencies. For instance, an image scaling operation which might be coded as:

\[
\text{Result} = \text{NewMin} + (\text{Image1} - \text{Min1}) \times \text{ScaleFactor};
\]

would typically require three image instructions, each having the inherent overhead of the loop control needed for their implementation.

Another situation where the usual coprocessor model results in inevitable inefficiencies arises with the use of template operators, where a hand coding could take advantage of the specific weight values to give a more efficient implementation.

Generating New Instructions Dynamically

If optimal efficiency is required then a traditional coprocessor model is not a solution. However, to retain the portability advantages of the coprocessor model while obtaining the efficiency of specific hand coded operations, EPIC is designed to be an extensible coprocessor, which can automatically generate new, optimised instructions to implement the compound operations actually occurring in the user program.

The heart of the EPIC approach involves several steps, which are made invisible to the application developer through the extensive use of operator overloading in C++:

- As library operations are called, a syntax tree is built. Upon assignment, if this is the first time occurrence of the tree, it will be executed (albeit inefficiently).
- Later, from this tree, optimised code for a new routine is generated (and compiled, etc.), to give a new instruction – thus extending the coprocessor’s instruction set.
- The next time this compound operation comes to be executed, the new, optimised instruction is retrieved and executed.

When observing the behaviour of an application program running in the EPIC environment, it will usually be noticed that the second time a program is run, it will do so faster than the first execution. The potential problem of encountering a non-optimised operation when running ‘live’ can be avoided by explicitly extracting such expressions, and defining them as functions separately.