Achieving Portability and Efficiency Through Automatic Optimisation: An Investigation in Parallel Image Processing

D Crookes[1], P J Morrow[2], T J Brown[1], G McAleese[2], D Roantree[2] and I T A Spence[1]

[1] Department of Computer Science, The Queen's University of Belfast, Belfast BT7 1NN, UK

[2] Department of Computing Science, University of Ulster at Coleraine, Coleraine, BT52 7EQ, UK

Abstract. This paper discusses the main achievements of the EPIC project, whose aim was to design a high level programming environment with an associated implementation for portable parallel image processing. The project was funded as part of the EPSRC Portable Software Tools for Parallel Architectures (PSTPA) programme. The paper summarises new portable programming abstractions for image processing, and outlines the automatically optimising implementation which achieves portability of application code and efficiency of implementation on a closely coupled distributed memory parallel system. The paper includes timings for optimised and unoptimised versions of typical image processing algorithms; it draws the main conclusion that it is possible to achieve portability with efficiency, for a specific application, by adopting a high level algebraic programming model, together with a transformation-based optimiser which reclaims the loss of efficiency which an algebraic approach traditionally entails.

1 Introduction

The need for portability of program code for parallel systems is an important but elusive goal. For instance, in an anecdotal keynote address to a recent European conference on parallel computing a distinguished American visitor from a major research facility recounted how the parallel supercomputers which his laboratory uses are renewed typically every three years. After installation, some two years is then spent modifying the suite of programs which researchers at the laboratory use so that they will run on the new hardware. Useful computing can then proceed for a period of about a year, before the machines are replaced again by the next generation; and so the cycle of events repeats. His purpose in relating these events was to highlight the relative difficulty in producing parallel software and porting it between machines. Software for parallel computers is often tailored to the architecture of particular machines in order to deliver optimal performance, particularly in the case of distributed memory machines.

On the other hand, this complicates the problem of constructing programs and inhibits their portability once constructed.

The EPIC project has been undertaken to consider to what extent automatic optimising software tools can take a high level, portable algorithm description, and generate parallel code whose efficiency on a distributed machine rivals than of hand-tuned parallel code. Because of the difficulty of such a problem in a general purpose programming environment, the project has deliberately chosen an application specific approach - in our case, the domain of image processing. The result is the EPIC environment: a high level, portable image processing programming environment, with an implementation which automatically generates very efficient code running on distributed memory parallel systems[1, 2].

More specifically, the key objectives of the EPIC project were:

1. To identify a set of programming abstractions for image processing. These abstractions constitute the core of the *Extensible Parallel Image Coprocessor* (EPIC) model. In the long term, we also wished to see how far these abstractions could be pushed towards general purpose programming abstractions.
2. To construct an (object-oriented) application development environment based on the EPIC model which enables users to construct portable image processing applications.
3. To develop a rule based transformation system which generates parallel code, for distributed memory machines, whose efficiency rivals that of hand tuned parallel code.

In the rest of this paper we first summarise the new image processing programming abstractions which were developed at the outset of the project, based on extensions to Image Algebra[3]. We then describe the EPIC application development environment in general, before looking at the off-line optimiser in particular. We present performance figures illustrating the benefit which optimisation brings for a number of simple algorithms and for the three architectures on which the system has now been implemented. Finally we draw conclusions and review the scientific insights which we have gained.

2 EPIC Programming Abstractions for Parallel Image Processing

It is known that efficiency on distributed memory parallel architectures benefits from the predictability and locality of data references. For low level image processing applications this is readily provided by the neighbourhood based operations which are typically used. For these forms of operation one can identify easily any data communication needs associated with an updating operation from the neighbourhood itself. Thus an application specific neighbourhood based programming model offers an advantage over more general purpose language notations where operations would typically be defined in terms of loop constructs and subscripting, from which the extraction of communication requirements is in general non-trivial.

Another advantage of a neighbourhood based programming model is that it can readily be supported by the development of a high level algebraic notation. This has been done for image processing by the development of Image Algebra[3] whose basic operations, plus image and template data types, form the starting point for the EPIC programming notation. Our programming abstractions for image processing have been developed in two stages:

1. Because *neighbourhood* processing is particularly suited to implementation on *parallel* machines (since data access patterns are predictable), the basic operations and concepts of *Image Algebra* were used as the basis of our programming abstractions, with minor extensions.
2. To continue to exploit the locality property of neighbourhood processing, we extended considerably the concept of a neighbourhood. For instance, we introduced *sets* and *sequences* of neighbourhoods. Together with variant neighbourhoods of Image Algebra, these novel abstractions give a very powerful and flexible notation which is still nevertheless capable of parallel implementation. As a result, we can now express various complete image transforms such as the Hadamard transform at a high level[4], and obtain an implementation whose performance is comparable with hand coding. We can also express various kinds of geometric transformation, including downsampling and upsampling operations of the kind used in some forms of wavelet transform, and data permutation operations such as perfect shuffle or the bit reversal permutation which is a component of many standard image transforms such as the FFT.

Evaluation of the abstractions was carried out by re-engineering a number of existing (sequential) systems. One large one (on calculating optical flow) has highlighted the need for extending the abstractions to include 3-D and video image processing applications. (The latter is currently the subject of more recent work at QUB[5].)

In trying to extend the applicability and expressive power of the above abstractions beyond the domain of image processing, we have insisted on retaining the basic concept of neighbourhood processing (as in (2) above). We have investigated several standard numerical algebra problems, and found that, with facilities for building new compound operators from a group of primitives, sets and sequences, we can code algorithms for problems such as matrix multiplication, LU factorisation, and finding the transitive closure of directed graphs. This work has not yet been reported in detail, but results to date indicate that a number of numerical problems *can* be expressed in terms of neighbourhood processing, but whether or not it is *natural* to do so needs further investigation.

3 Implementation and the Need for Optimisation

Although the kind of high level, algebraic notation outlined above is very convenient as a programming notation, there are some forms of inefficiency which necessarily arise from a straightforward library-based implementation approach. There are two main reasons for this:

- The fact that *special cases*, in which an operands value is known and could be exploited manually, cannot be fully exploited, because the implementations of the high level operations must of necessity be general purpose.
- The fact that implementing expressions involving *compound* operations will frequently involve replication of overheads.

As a simple example of the first problem, consider a convolution operation between an image and a template (a weighted window used in neighbourhood operators). This operation is normally provided as a general purpose routine applicable to any pair of operands. However it is frequently the case that the template may contain weights of zero or one. Using a general purpose routine will therefore entail unnecessary arithmetic operations.

The second problem arises with any expression which involves compound operations using image operands. Each individual operation will normally be implemented by a routine which traverses the image domain (for example using a double loop construct). When a compound expression is implemented this overhead is replicated for each individual operation, leading to a significant loss in efficiency.

Of course, a programmer interested primarily in efficiency would manually write additional routines for all the above cases; but this would require operating at a lower level at which the parallelism and communication would be visible, thus reducing portability. The essence of the EPIC environment is that it automatically generates these additional routines, and links them in to the runtime environment; in this way it gives the same efficiency as a good programmer, but allows the application developer to continue to operate at the highest level. The environment is based on a rule based optimiser which aims to apply the same reasoning steps which a programmer would apply manually in generating efficient, tailored versions of the standard routines.

4 EPIC : A Portable Application Development Environment

The second objective defined above was the implementation of an object oriented application development environment for parallel image processing based on the programming abstractions defined within the framework of the EPIC model. We have developed a sophisticated environment, providing C++ as the users language, with the EPIC abstract machine provided as a range of C++ methods[6, 7]. In developing the EPIC environment, we have proposed and integrated several ideas which we believe could have longer term benefits for the design of future systems of this kind. The more significant ideas (or developments of previous ideas and approaches) which appear to us to have particular relevance and importance in this area of the project are now considered.

An Extensible Abstract Machine

The traditional approach to achieving portability across architectures is to define an abstract machine with an instruction set matching the application. In our case, we provide an image coprocessor with an instruction set which supports our Image Algebra-based programming abstractions. This results in a static instruction set (like a library). However this traditional approach results in inefficiencies which can sometimes make the whole approach less attractive to real application developers. As discussed above, these inefficiencies arise typically for two reasons:

1. Special cases of instructions are often implemented more efficiently by manual programmers, and
2. The array processing overheads for compound instructions are usually cumulative.

We have developed an architecture for an abstract machine in which these inefficiencies can be avoided while retaining the elegance and portability of the abstract machine model. We have proposed the concept of an Extensible abstract machine, in which the instruction set has two parts:

1. The basic, static instruction set, and
2. A set of additional, optimised instructions which avoid the inefficiencies mentioned above.

These additional instructions are program-specific, and implement special cases and compound instructions as efficiently as manual coding. As indicated below, these extended instructions are generated and called automatically.

A Self-Optimising Abstract Machine

To enable the programmer to continue to use the static EPIC algebraic programming abstractions, and thus gain clarity, portability and conciseness, while at the same time gain the performance benefits of the types of optimisation mentioned above, we have developed an off-line optimiser which carries out the optimisation and generation task which a performance-minded programmer would carry out. The EPIC environment automatically builds new, optimised instructions which are then linked in to the dynamic, extended part of the EPIC instruction set. This involves the following program execution behaviour:

1. The first time a program is run, it does so using only the static instruction set. But at the same time, syntax trees for compound operations and special cases are dumped to file.
2. Off-line, these syntax trees are transformed into new, optimised routines, and linked into the EPIC coprocessors extended instruction set.
3. On subsequent execution, the extended instructions are automatically used, with no user input. The only visible difference will be the increase in execution speed.

To demonstrate how these ideas are provided by the EPIC environment, the next section presents the architecture of the EPIC system itself.

The EPIC System Architecture

The EPIC application development environment has three principal components. These are:

1. The C++ class library which forms the users application programming environment.
2. The parallel coprocessor which is installed on the parallel machine.
3. The off-line optimiser which is the rule-based transformation system used to generate extended instructions from combinations of the basic programming abstractions. The optimiser is described in more detail in the next section of this report, but it forms an integral part of the overall system architecture.

The overall system architecture is illustrated in Figure 1 below.

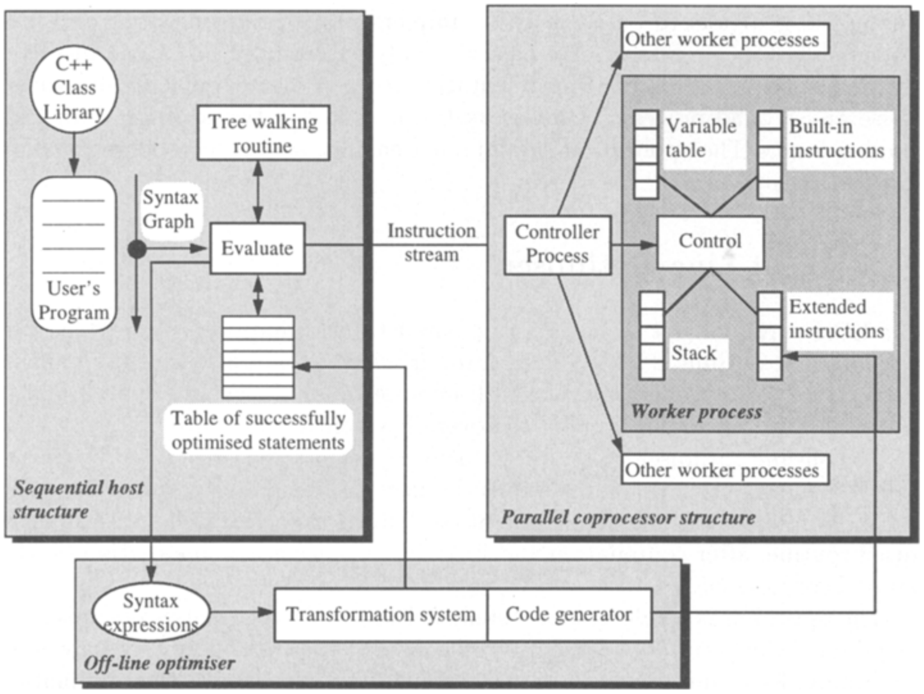

Fig. 1. Overall System Architecture

The Parallel Coprocessor

The parallel coprocessor consists of a controller process and a set of worker processes. Images are geometrically decomposed into horizontal strips or vertical columns. Dynamic redistribution at runtime is supported.

The worker processes perform the actual computations on image data. Each worker process has the internal structure illustrated in the inner box (Figure 1), with a control module, a variable table storing information about program variables including pointers to their data, an evaluation stack, and two tables of instructions. The first is the table of built-in instructions, and the second is the table of extended instructions generated by the optimiser. The control logic can handle the implementation of either form of instruction.

The coprocessor is implemented in C, using MPI for communications. In fact, only six MPI routines are used. On the C40 system, we wrote our own MPI routines, making sure they implement only the minimum necessary functionality (which proved a significant performance advantage).

Portability of the System

Our initial goal was to base the EPIC implementation on a network of Texas Instruments C40 processors. We have not only successfully met this objective, but we have also ported the implementation to two other parallel architectures. These are: a quad-processor Unix workstation, and a network of Pentium PCs running Linux. The application development environment is fully operational on all three architectures.

5 The Off-Line Optimiser

The EPIC environment includes a rule-based transformation system which can generate new optimised instructions from arbitrary compositions of the built-in instructions. This component is the off-line optimiser shown in Figure 1 and it is in reality an integral feature of the overall system architecture.

The off-line optimiser takes the syntactic representations generated by execution of the users program on the sequential host and generates optimised parallel code to implement the complete operation defined in each statement. The optimised routine, after compilation and linking, becomes a new instruction on the parallel coprocessor.

The optimiser is a rule driven system written in Prolog. It uses optimisation techniques which are not in themselves new. For the most part these are loop combining, loop unrolling and loop interchanging, along with the elimination of redundant arithmetic. Several strategies are employed to identify the most appropriate path to follow based on the contents of the syntax graph. This includes trial and error approaches when an obvious course is not evident from the nature of the syntax graph. A pattern-matching approach is used to find the best transformation applicable.

The output is initially in the form of an object code syntax graph. A code generation module is then used to generate C code with calls to library routines for such purposes as border swapping. The generated optimised instructions are independent of the number of available processors. Direct calls to MPI, which provides the communication framework, are also embedded in the code.

The capabilities built into the optimiser enable it to handle any composition of the operations provided within the class library, including operations nested to any depth. Also provided is the ability to handle the syntactic representations of variant templates operations, including sequences of variant templates.

6 Performance of the EPIC Optimiser

As an illustrative case, consider the example program statement of the form:

```
EdgeImage = (absconv(Im,Sv) + absconv(Im,Sh)) > Threshold)*255 ;
```

This performs a simple edge detection operation when `Sv` and `Sh` are vertical and horizontal gradient operators such as the Sobel operators. When we execute a statement like this without optimisation a total of 5 of the basic instructions will be needed, each entailing a pass over the image domain. Optimisation can produce a new instruction requiring only one traverse of the image domain to perform the whole operation. Our initial studies showed that a sequential unoptimised version would run up to nearly 5 times slower than a hand- optimised version for statements of this level of complexity (and a factor of around 4.5 on a parallel system).

Trials using the EPIC optimiser have now been performed on all three of the architectures on which the EPIC system is implemented. The results are tabulated (Table 1) for two such operations, namely a top hat filter operation and the edge detector example shown above. Times are in seconds and the image size in each case is 256*256.

Table 1. Execution times

	Top Hat Filter	*Edge Detector*
C40 - 4 Workers		
Before optimisation	0.62	0.39
After Optimisation	0.13	0.10
Improvement	4.8 fold	3.9 fold
C40 - 2 Workers		
Before optimisation	1.12	0.47
After Optimisation	0.17	0.16
Improvement	6.6 fold	2.93 fold
PC-Linux System		
Before optimisation	1.83	0.94
After Optimisation	0.75	0.53
Improvement	2.44 fold	1.77 fold
Quad-Processor Workstation		
Before optimisation	2.87	1.95
After Optimisation	1.74	1.36
Improvement	1.65 fold	1.43 fold

These are of course quite short programs but they do illustrate that on the C40 implementation we are seeing impressive performance improvements from the optimisation.

Measurement of the overheads incurred by the host processor (tree building, tree matching and host-coprocessor communication) is harder to assess. However, on a selection of tests with a (slow!) SUN host and a C40-based coprocessor, the host overheads reduced overall system performance by between 10% and 35%. This demonstrates another significant finding namely, the importance of a fast, closely-coupled host to a parallel coprocessor.

It is clear from all our experiments that workstation clusters are going to struggle in obtaining parallel efficiency for image processing, because of the communication overheads and process switching times. On such systems, the EPIC optimisation capability will probably not give sufficient speedup to make its use worthwhile (although recent work on lightweight messaging[8] could improve the situation somewhat). However, the C40 system gives very promising results, and the EPIC approach definitely pays off. This is largely because the C40 system is much more closely coupled, and because implementation of the MPI communication routines are architecture-aware. This is an important lesson for other implementors.

7 Conclusions

The EPIC project has demonstrated the main thesis of our approach to achieving portability with efficiency, namely that in this problem domain:

1. Portability over parallel architectures can be achieved by adopting an algebraic, application-specific programming model, and by retaining the traditional implementation concept of an abstract coprocessor machine - but in a more sophisticated form.
2. The inherent loss of efficiency arising from a standard abstract machine-based approach to implementing an algebraic model can be recovered, by developing a rule-based optimiser which generates the equivalent of new machine instructions dynamically, thus giving an extensible instruction set for the abstract coprocessor.

As part of the project, we have therefore developed a powerful self-optimising tool which demonstrates the feasibility of automatically generating new operations from special cases or compositions of library operations, and have provided a transformation system for program optimisation. An efficient implementation of EPIC has been developed for C40 networks and portability has been achieved through the use of an MPI communications layer. In addition the system has been ported to a quad-processor Unix workstation and a PC-based Linux network.

The work described in this paper has contributed to the ongoing debate on how to achieve portability with efficiency, though very much from an application specific viewpoint. The main contributions to this debate are:

- The concept of an *extensible* abstract machine, for obtaining both portability and efficiency.
- New sophisticated *neighbourhood-based abstractions*, which are proving capable of expressing some algorithms from other application areas (e.g. numerical problems), and which guarantee parallel efficiency.
- Automatic *identification* of extended instructions has been achieved, though we are undecided at this stage as to whether or not it is *advisable* (the programmer could identify these manually).
- The novel concept of a *self-optimising machine*, made possible by automatic extension of the instruction set by the system generating, linking and calling extended instructions on subsequent program executions.
- Portability across different distributed memory platforms, using a small subset of the MPI interface. For full efficiency, we recommend rewriting these MPI routines for a specific configuration, rather than rely on a general-purpose MPI implementation.
- For image processing applications at least, the benefits of automatic optimisation are most apparent on a *closely coupled network* of processors. They are not so apparent on a loosely coupled cluster with a general purpose MPI implementation.
- At the highest level, possibly our most significant achievement is to enable programmers to continue to operate using a set of high level primitives — and hence retain application *portability* — by reclaiming the usual inherent loss of efficiency through sophisticated optimising software tools. It is this concept which should be applicable to a wide range of other application domains.

Acknowledgements

This work has been supported by the UK EPSRC within the framework of the PSTPA (Portable Software Tools for Parallel Architectures) initiative. The support received from Transtech Parallel Systems is also gratefully acknowledged.

References

1. Crookes, D., Brown, J., Dong, Y., McAleese, G.,Morrow, P.J., Roantree D. and Spence I.: A Self-Optimising Coprocessor Model for Portable Parallel Image Processing. Proc. EUROPAR'96. Springer-Verlag (1996) 213–216
2. Crookes, D., Brown, J., Spence, I., Morrow, P., Roantree, D. and McAleese, G.: An Efficient, Portable Software Platform for Parallel Image Processing. Proc. PDP'98, Madrid (1998)
3. Ritter, G.X., Wilson, J.N. and Davidson, J.L.: Image Algebra : An overview. Computer Vision, Graphics and Image Processing **49** (1990) 297–331
4. Crookes, D., Spence, I.T.A. and Brown, T.J.: Efficient parallel image transforms: a very high level approach. Proc. 1995 World Transputer Congress, IOS Press (1995) 135-143

5. Hill, S. J., Crookes, D. and Bouridane, A.: Abstractions for 3-D and video processing. Proc. IMVIP-97, Londonderry, (1997)
6. Morrow, P.J, Crookes, D., Brown, J., Dong, Y., McAleese, G., Roantree, D. and Spence, I.: Achieving Scalability, Portability and Efficiency in a High-Level Programming Model for Parallel Architectures. Proc. UKPAR'96. Springer-Verlag (1996) 29–39
7. Morrow, P., Roantree, D., McAleese, G., Crookes, D., Spence, I., Brown, J. and Dong, Y.: A Portable Coprocessor Model for Parallel Image Processing. European Parallel Tools Meeting. ONERA, Chatillon, France (1996)
8. Nicole, D.A. et al. High performance message passing under chorus/Mix using Java. Department of Electronics and Computer Science, University of Southampton (1997)