Using eSkel to implement the multiple baseline stereo application

Anne Benoit[a], Murray Cole[a], Stephen Gilmore[a], Jane Hillston[a]

[a]School of Informatics, The University of Edinburgh, James Clerk Maxwell Building, The King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK

We present an application of the *eSkel* skeletal programming library to the multiple baseline stereo problem. We compare its performance to that of a direct MPI implementation of the same algorithm.

1. Introduction

The skeletal approach to parallel programming is well documented in the research literature (see [4,5,7,8] for surveys). It observes that many parallel algorithms can be characterised and classified by their adherence to one or more of a number of generic patterns of computation and interaction. Skeletal programming proposes that such patterns be abstracted and provided as a programmer's toolkit, with specifications which transcend architectural variations but implementations which recognise these to enhance performance. This level of abstraction makes it easier for the programmer to experiment with a variety of parallel structurings for a given application, by enabling a clean separation between structural aspects and application specific details. In the *eSkel* (Edinburgh Skeleton Library) project, motivated by our observations [5] on previous attempts to implement these ideas, we have begun to define a generic set of skeletons as a library of C functions on top of MPI.

This paper describes the first use of *eSkel* on a significant application, the multiple baseline stereo vision problem [6,10]. We begin by providing an overview of *eSkel* and its conceptual basis, before proceeding to a description of the standard multiple baseline stereo algorithm. We describe the facility with which the algorithm can be expressed in *eSkel*, and examine the performance of the resulting programs on a Beowulf cluster and on an SMP, focusing particularly on the overhead incurred by *eSkel* when compared with an explicit MPI version of the same algorithm.

2. Structured parallel programming with *eSkel*

The *eSkel* project [1–3] is an ongoing attempt to investigate the practicality and applicability of skeletal programming systems. Building on previous experiences, its aims (to which we will return in our experimental evaluation) were stated [5] as being to develop a skeletal system which

1. Promotes skeletal programming with minimal **disruption** to the "conventional" parallel programmer's conceptual model.

2. Allows the integration of **ad-hoc** (unstructured) code within an otherwise skeletal program.

3. Accommodates a **flexible** collection of variations on familiar parallel programming idioms.

4. Provides empirical evidence that skeletal programming need not incur significant **performance** penalties (and indeed might even provide performance improvements) judged against equivalent ad-hoc parallel programs.

eSkel's conceptual model and API are based around MPI. This basis, and the fact that the implementation is also built on top of MPI, ensure widespread portability.

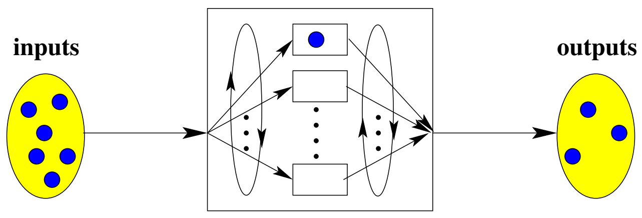

Figure 1. Conceptual structure of the *Deal* skeleton.

In essence, an *eSkel* skeleton is a collective operation, called by all processes within a given MPI communicator group. The call causes the processes to take on the roles of the various components of the chosen skeleton (for example, the stages in a pipeline). It abstracts the implied communications, either completely, leaving the programmer to specify only the activities within each component, or partially, allowing the programmer to control the timing of communications within the spatial constraints imposed by the skeleton (for example, allowing a pipeline stage to consume an input without producing a corresponding output).

Skeleton calls may be nested, either *transiently*, meaning that the inner instantiation is created and exists only between interactions of the outer call, or *persistently*, meaning that the inner call lasts for the duration of the entire outer call (in effect giving a flat skeleton structure combining the two skeletons).

In the application described here, we use the only two of *eSkel*'s collection of skeletons which have so far been fully implemented, the *Pipeline* and the *Deal*. Our pipeline is a straightforward abstraction of the familiar paradigm: a sequence of *stages*, any of which may be internally parallel, processes a sequence of input items to produce a sequence of output items. *Deal*, while less familiar by name, captures another familiar technique, typically applied within bottleneck pipeline stages: the stage computation itself is replicated, with successive inputs dispatched cyclically to the replicas, outputs being merged in the original order into the overall stream passed to the subsequent stage. The technique is only applicable for stages in which no internal state is maintained from one input to the next. Figure 1 sketches the structure of a single *Deal*. More typically, either or both of the input and output streams will be tied to other pipeline stages. Thus, the *Deal* skeleton behaves like a *Farm* with a cyclic allocation of the work to the farmers.

A full discussion of the *eSkel* API is well beyond the space constraints of this paper.

3. The multiple baseline stereo application

The multiple baseline stereo problem [6,9,10] involves measuring depth in a scene with the help of several cameras. The cameras have different baselines, enabling precise distance estimates to be obtained with a stereo matching method. Paraphrasing [9],

> "The input consists of three images, acquired from three horizontally aligned, equally spaced cameras. One image is the *reference image*, the other two are *match images*. For each of 16 disparities $d = 0..15$, the first match image is shifted by d pixels, and the second image is shifted by $2d$ pixels. A *difference image* is formed by computing the sum of squared differences between the corresponding pixels of the reference image and the shifted match images. Next, an *error image* is obtained by replacing each pixel in the difference image with the sum of the pixels in a surrounding 13x13 window. Finally,

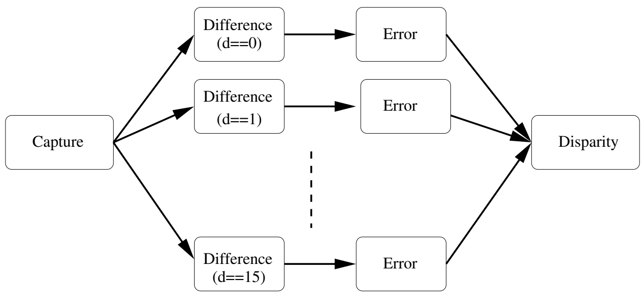

Figure 2. Pipelined structure of the original algorithm.

the *disparity image* is formed by finding, for each pixel, the disparity that minimises error. We then know the depth of each pixel, which depends on this disparity."

The algorithm naturally lends itself to pipelined and replicated parallelism, with stages for data gathering (or in our case, in the absence of cameras, artificial generation), difference image calculation, error image calculation and disparity image calculation. Each input to the pipeline is a set of three images. For a given triple of images, the difference and error calculations for each of the 16 disparities are independent and so there is scope for concurrent execution, as suggested by figure 2. Finally, we note that the algorithm could apply internal data parallelism to the difference and error computations.

The natural context of the application is in real-time processing of a stream of images. This means that the most appropriate measure of performance is the *throughput* with which such a sequence is processed. We adopt this metric in our performance graphs.

4. Parallel implementation with *eSkel*

For the purposes of our experiments we have chosen to implement a variant of the algorithm described above. As sketched in figure 3, we have built a three stage pipeline, in which the first stage generates images, the second implements all (16 per image set) of the difference, error and disparity computations, and the final stage (which might be a display operation in a real application), simply performs some checking of results. Our artificial inputs are constructed with easy checking of outputs in mind. As noted above, the run-time of the calculations, which is our main concern, depends only upon the size (rather than the content) of the images. Not surprisingly, this structure results in a substantial bottleneck in the second stage. We resolve this with a *Deal* - successive image sets are distributed to successive workers. We ran tests with between one and six replicas.

Stripped of application specific code, the program itself is straightforward. Figure 4 presents the main program (with only simple C declarations omitted to save space). Since images are generated within stage 0, and outputs are not stored, most of the data parameters are redundant (hence all the `NULL`s and `0`s). The `imodes`, `spreads` and `types` parameters to the skeleton call describe the structure of the interfaces between stages.

In this version of the program, we have chosen to use *explicit* communication for data leaving stage 0 and entering stage 2 (line 3). By way of contrast, the workers in the *Deal* will experience these communications in implicit mode. This is specified in two steps. For the stage itself, the interaction mode is given as "devolved", indicating the presence of a persistently nested skeleton.

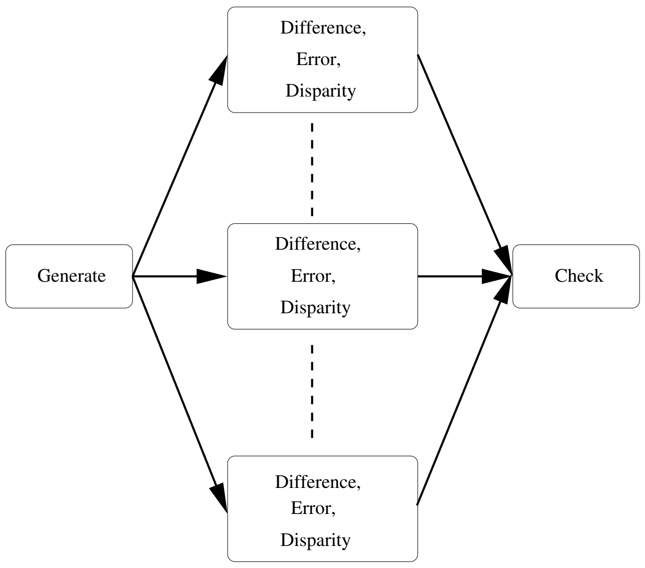

Figure 3. Structure of the algorithm as implemented.

An extract of the code for stage 2 (figure 5) completes the process, by specifying (line 4) *implicit* interaction mode.

Lines 17-19 of figure 5 are the interface between *eSkel*'s data model and the standard C of the computational fragments. Following the MPI approach, the three data arrays are received packed into a single array, accessible through the `data` field of the `eSkel_molecule_t` received by the stage. Rather than wasting time explicitly unpacking the structure, we simply create convenient pointers to the sections corresponding to each array. Lines 32-33 similarly prepare the result for transmission to the final stage. The bulk of the processing is done by the function calls on lines 28-29. These are written in standard C.

4.1. Performance

In assessing the performance of a new programming model (in our case, *eSkel*), it is important to try to distinguish overheads introduced by the implementation mechanism from constraints which are inherent characteristics of the algorithm or underlying machine. To this end, the following programs were written:

- **Sequential**. This represents a "sensible" rendition of the algorithm. The images are processed in sequence by a loop and there is no unnecessary copying, as might be performed by a sequential emulation of the parallel algorithm. No attempts have been made to optimise lower level aspects (e.g. thinking about array access patterns to enhance cache utilisation), but this is equally true of our parallel versions.

- **Raw MPI**. A straightforward MPI implementation of the adapted algorithm, generalised to allow for cyclic distribution of image sets to a number of workers in the second stage (in other words, a hand-coded "deal", specialised to this application, and thereby omitting some of the data marshalling hidden inside the *eSkel* versions).

- **eSkel**. The "*Pipeline* and *Deal*" program described above.

```
  spread_t spreads[STAGES+1]   = {SPGLOBAL, SPGLOBAL,SPGLOBAL,SPGLOBAL};
  MPI_Datatype types[STAGES+1] = {MPI_INT, MPI_INT, MPI_INT, MPI_INT};
  Imode_t imodes[STAGES]       = {EXPL, DEV, EXPL};
  eSkel_molecule_t *(*stages[STAGES])(eSkel_molecule_t *) = {
    (eSkel_molecule_t * (*)(eSkel_molecule_t *)) stage1,
    (eSkel_molecule_t * (*)(eSkel_molecule_t *)) stage2,
    (eSkel_molecule_t * (*)(eSkel_molecule_t *)) stage3
  };

  MPI_Init(&argc, &argv);
  SkelLibInit();

  MPI_Type_contiguous((ROWS+WINY)*COLS, MPI_INT, &MPI_int_array);
  MPI_Type_commit(&MPI_int_array);
  types[1]   = MPI_int_array;  types[2]   = MPI_int_array;

  if (myrank()==0)                mystagenum = 0;
  else if (myrank()==nprocs-1)    mystagenum = 2;
  else                            mystagenum = 1;

  Pipeline (STAGES,  imodes, stages, mystagenum, BUF, spreads, types,
            NULL, 0, 0, NULL, 0, &outmul, 0, mycomm());

  MPI_Finalize();
```

Figure 4. The main program

```
void stage2 (void) {
  int outmul;

  Deal (mycommsize(), IMPL, worker, myrank(), STRM,
        NULL, 0, 0, SPGLOBAL, MPI_int_array,
        NULL, 0, &outmul, SPGLOBAL, MPI_int_array, 0, mycomm());
}

eSkel_molecule_t * worker (eSkel_molecule_t *item) {
  int *ref, *m1, *m2, i, j;

  float curbesterr[ROWS*COLS];
  int   curbestdisp[ROWS*COLS];
  float diffimg[(ROWS+WINY)*COLS];


  ref           =  &((int *)(item->data[0]))[0];
  m1            =  &((int *)(item->data[0]))[(ROWS+WINY)*COLS];
  m2            =  &((int *)(item->data[0]))[2*(ROWS+WINY)*COLS];

  for (i=0;i<ROWS;i++)
    for (j=0;j<COLS;j++) {
        curbesterr[i*COLS+j]  = 9999999.0;
        curbestdisp[i*COLS+j] = 0;
    }

  for (curdisp=0;curdisp<MAXDISP;curdisp++) {
    gendiffimg(ref,m1,m2,diffimg,curdisp);
    updatedispimg(diffimg,curbesterr,curbestdisp,curdisp);
  }

  item->len[0]  = 1;
  item->data[0] = curbestdisp;
  return item;
}
```

Figure 5. Code for stage 2.

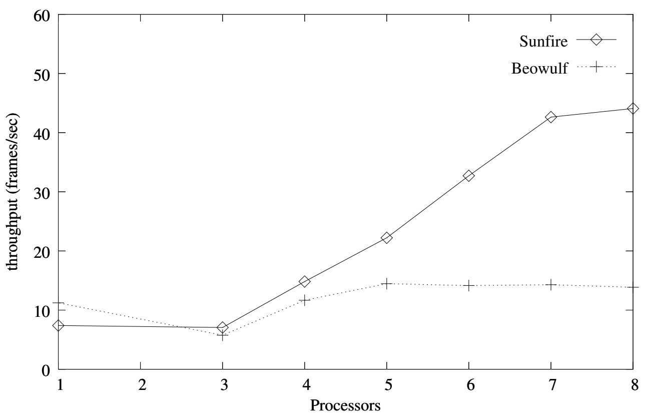

Figure 6. Performance of the raw MPI algorithm on the two machines.

Our experiments were conducted on two platforms:

- A 64 node (1.8 GHz Intel Pentium 4 processors) Beowulf cluster networked with a 100Mb ethernet, using the Los Alamos Message Passing Interface LA-MPI.
- A 52 node (0.9 GHz Ultrasparc III processors) Sunfire E15k SMP with the native Sun implementation of MPI.

All runs were repeated six times, with results shown below being averages. None of the runs deviated from the average to any interesting extent. All experiments involved the processing of 20 images, each of 240×256 pixels. We measure performance in terms of throughput, as described above.

4.2. Performance of the underlying algorithm

Before assessing *eSkel*, it is important to understand the capabilities of the parallelised multiple baseline stereo algorithm itself. Figure 6 compares the performance of the sequential and raw MPI programs on each of the platforms. The sequential performance (data points for one processor) is a little better on the Beowulf, presumably reflecting the greater speed of its processors. There are no data points for two processors because our chosen parallelisation requires at least three processors. For four and more processors, we see the effect of adding extra workers to the second stage: the data points for p processors indicate the use of a directly programmed "deal" with $p - 2$ workers. We notice that performance on the Beowulf is disappointing, with the parallel versions only eventually marginally beating the sequential one, and with the curve flattening very quickly. This suggests that the relatively low computation to communication ratio inherent in the algorithm is too much for this machine to cope with. In contrast, the situation on the SunFire is much more promising, with parallelisation apparently worthwhile, (flattening at about eight processors).

4.3. Performance of *eSkel*

We can now investigate the performance of *eSkel*. Figure 7 compares the throughput of the raw MPI and corresponding *eSkel* programs on the Beowulf. The first data point is for three processors because, as noted above, this is the smallest natural number of processors for a parallel version. We observe that the two programs perform quite similarly, with a performance loss of around 8% for *eSkel*. Figure 8 illustrates a similar comparison for the Sunfire. The overall pattern is similar, with a performance gap which is slightly wider (around 10-13%). We are encouraged by these results, particularly since we have already observed that the application is inherently communications-heavy,

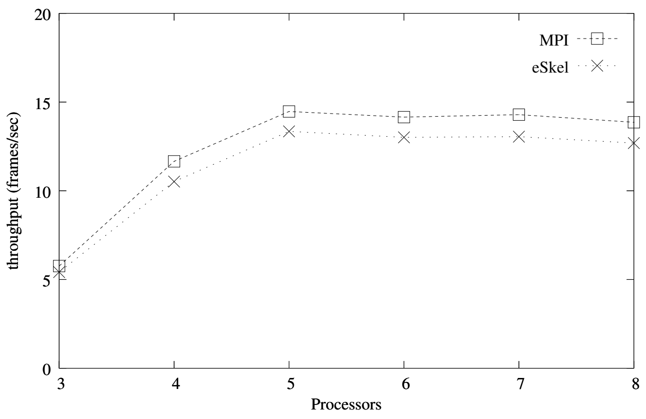

Figure 7. Performance of *eSkel* and MPI on the Beowulf.

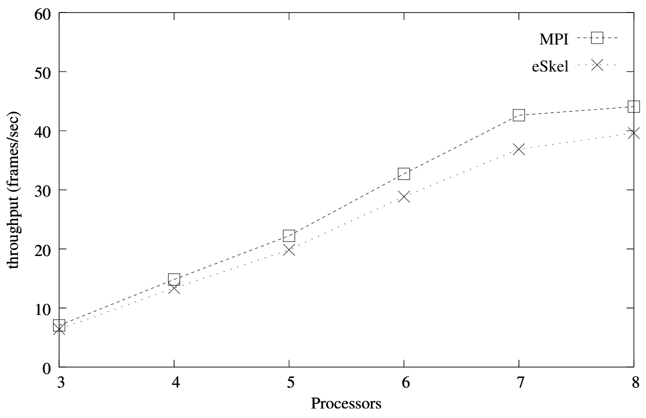

Figure 8. Performance of *eSkel* and MPI on the Sunfire.

and so can be expected to highlight any weaknesses in the underlying system. We additionally note that the current implementation of *eSkel* is a proof-of-concept prototype, with considerable scope for optimisation, and expect that the performance gap highlighted here can be significantly closed.

The observant reader may wonder why the performance gap is more noticeable for the Sunfire, when intuition might suggest that this machine, with its relatively superior communications technology, might be more forgiving of communications profligacy. One credible hypothesis might suggest that this is related to the relative merits of the implementations of MPI on the two machines: *eSkel* makes heavy use of collective communications (particularly `MPI_Scatterv` and `MPI_Gatherv`). If these operations were less effectively implemented (with respect to simple sends and receives) on the SunFire than on the Beowulf, then we would expect the observed effect. However, simple experiments with the operations were unable to produce any convincing evidence. Another hypothesis would suggest that there may be a cache-related effect at work. Remembering that all "communication" in the Sunfire is underpinned by shared memory, might it be the case that the sends and receives of the raw MPI versions have better cache side-effects than the scatters and gathers of the *eSkel* implementations?

5. Conclusions

We conclude with some observations motivated by the assessment criteria laid out in section 2.

1. **Conceptual disruption**. The "skeletons as collective operations" approach seemed to fit the algorithm neatly. The pointer twiddling and casting to interface to incoming and outgoing data was messy. While this is partly inherited from MPI (the raw program needed similar code to allow the arrays to be transmitted efficiently), the molecule concept added a level of indirection.

2. **Ad-hoc parallelism**. The application itself is too regular to need this, but the performance monitoring code (omitted from the extracts here) exploited this facility.

3. **Flexible skeletons**. It was helpful to be able to program a pipeline in which the data stream was generated directly by the first stage.

4. **Performance**. As noted above, this seems to be at least encouraging.

6. Acknowledgements

The authors are supported by the Enhance project ("Enhancing the Performance Predictability of Grid Applications with Patterns and Process Algebras") funded by the Engineering and Physical Sciences Research Council under grant number GR/S21717/01.

References

[1] A. Benoit and M. Cole. *eSkel's web page*. `http://homepages.inf.ed.ac.uk/mic/eSkel`.

[2] A. Benoit and M. Cole. Two Fundamental Concepts in Skeletal Parallel Programming. In V. Sunderam, D. van Albada, P. Sloot and J. Dongarra (eds), *International Conference on Computational Science (ICCS), Part II*, LNCS 3515(764–771), Springer, 2005.

[3] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible Skeletal Programming with eSkel. *To appear in Proceedings of EuroPar 2005*, LNCS, Springer, 2005.

[4] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press & Pitman, ISBN 0-262-53086-4, 1989.

[5] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[6] M. Okutomi and T. Kanade. A Multiple-Baseline Stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(4):353–363, April 1993.

[7] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, ISBN 0-7484-0759-6, London, 1998.

[8] F.A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer Verlag, ISBN 1-85233-506-8, 2003.

[9] J. Subhlok, D. O'Hallaron, T. Gross, P. Dinda, and J. Webb. Communication and memory requirements as the basis for mapping task and data parallel programs. In *Proceedings of Supercomputing '94*, pages 330–339, Washington, DC, November 1994.

[10] J.A. Webb. Latency and Bandwidth Considerations in Parallel Robotics Image Processing. In *Supercomputing'93*, pages 230–239, Portland, OR, November 1993.