Introduction of Static Load Balancing in Incremental Parallel Programming

Joy Goodman and John O'Donnell

Glasgow University

Abstract. Formal program transformation in a functional language can be used to support incremental design of parallel programs. This paper illustrates the method with a detailed example: a program transformation that improves the static load balance of a simple data parallel program.

1 Introduction

Incremental design of programs allows the critical design decisions to be introduced one at a time, in a logical sequence. Sometimes this is simpler than writing the program directly in its final form, where all interconnected design decisions are made simultaneously. Incremental design is particularly useful for parallel programming [2,1], where there is a large space of decisions (eg. how to employ the available processors, how to distribute the data). One successful incremental methodology for parallel programming with a mixture of task and data parallelism is the TwoL model [4].

Incremental programming fits well with pure functional languages, such as Haskell, which provide a sound foundation for correctness-preserving program transformation [3]. Each step in the program derivation, which introduces additional low level detail about the implementation, can be proved equivalent to the preceding version of the program. There is also the potential for formal correctness proofs and support by automated tools.

This paper investigates the use of Haskell for incremental parallel programming based on formal transformations. In particular, we focus on one aspect of the design: the use of program transformations to improve the static load balance of a program, using a simple cost model to guide the transformation.

2 Expressing the Parallelism

The program is expressed using a *coordination language* that specifies how the computation may be performed using a sequence of *operations*, which may be abstract about the parallelism, or sequential or parallel. We use a superscript S or P to indicate that an operation is definitely sequential or parallel, and no superscript when this has not yet been decided.

The operations express computations over data structures called *finite sequences*, with types of the form $FinSeq\,\alpha$. Variants on this are $ParFinSeq\,\alpha$,

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 535–539, 2001.

whose elements are distributed, one per processor, and *SeqFinSeq* α, where all the elements reside in the same processor. A typical operation is

$$map \;::\; (a \to b) \to \mathit{FinSeq}\ a \to \mathit{FinSeq}\ b,$$

which applies a function to each element of its argument. A sequential map, map^S, iterates the function over data stored in the memory of one processor, while a parallel map, map^P, can apply the function to multiple data elements simultaneously since they are stored in different processors.

In the remainder of this paper we will use as a running example the reduction of the columns of a lower triangular matrix X (Figure 1(a)), using an operator $\oplus$, which we assume to be a relatively expensive computation. The matrix contains a sequence of n columns $X_i = [x_{0,i},\ \ldots,\ x_{n-1,i}]$ for $0 \le i < n$, where column i contains $i + 1$ elements. The aim is to compute the vector of column sums: $s_i = \bigoplus_{j=0}^{n-1} x_{j,i}$ for $0 \le i < n$.

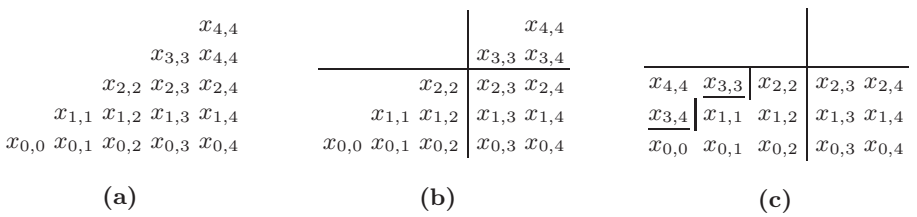

Fig. 1. **(a)** Original matrix; **(b)** partitioning; **(c)** load-balanced computations

This mathematical specification can be expressed in Haskell, with f in place of $\oplus$, by representing the matrix as a finite sequence of columns. Each column is a finite sequence of data values of type α. The columns are reduced using *foldl*, which takes an extra parameter, a, a unit of f. The unit parameter is not necessary at this point, but it simplifies the following presentation of the transformation. Each column of the matrix is reduced in a dedicated processor, so the map is parallel, but the reductions (folds) are performed sequentially. This is a more concrete version of the specification which would be written using *FinSeq* α.

$$\begin{aligned} &\mathit{maptri} \;::\; (\alpha \to \alpha \to \alpha) \to\ \alpha\ \to \mathit{ParFinSeq}\ (\mathit{SeqFinSeq}\ \alpha)\ \to\ \mathit{ParFinSeq}\ \alpha \\ &\mathit{maptri}\ f\ a\ \mathit{xss}\ =\ \mathit{map}^P\ (\mathit{foldl}^S\ f\ a)\ \mathit{xss} \end{aligned}$$

3 Generic Load Balancing

During the incremental derivation of a parallel program, we gradually transform it, in order to bring it closer to the low level executable code while improving its

efficiency. One potential optimisation is *static load balancing*: reorganising the program so that the workload is spread evenly across the available processors. The *maptri* example is well suited for static load balancing, and this technique is useful for many realistic applications. Many problems, however, have irregular or dynamic processor loads that render static load balancing ineffective. In such cases, a better solution is *dynamic load balancing* supported by the runtime system.

The static load balancing is guided by a cost analysis that takes account of both the computation and communication costs. We illustrate the method by introducing a simple cost model. Let T_f be the time needed for a processor to apply f to an argument stored in the local memory, assuming that T_f does not depend on the value of the argument. Let $T_{com} = T_0 + k \cdot T_c$ be the time required by a total exchange operation, where k is the size of the largest message.

In the *maptri* program, processor i requires time $T\ (foldl^S\ f\ a\ [x_{0,i},\ \ldots,\ x_{i,i}]) = i\ \cdot\ T_f$. This implies a poor load balance, since the processors' computation times vary from 0 to $(n-1) \cdot T_f$, and the time for the whole program depends on the maximum time required by a processor. By spreading the work evenly the computation time could be cut in half, although we must also consider the costs of the communications introduced to balance the load.

The first step in load balancing is to divide the tasks up into smaller pieces, and a natural idea is to split the folds over a long list into separate folds over shorter pieces (Figure 1(b)). The partial folds can then be rearranged so that each processor has about the same amount of work (Figure 1(c)). The following lemma permits the splitting of folds, provided that f is associative:

Lemma 1. *If* $f :: \alpha \to \alpha \to \alpha$ *is associative, and* $xs,\ ys\ ::\ SeqFinSeq\ \alpha$, *then* $foldl^S\ f\ a\ (xs +\!\!+\ ys)\ =\ f\ (foldl^S\ f\ a\ xs)\ (foldl^S\ f\ a\ ys)$.

Now we have to divide up each of the columns. This can be done using the following lemma.

Lemma 2. $xs\ =\ take^S\ m\ xs\ +\!\!+\ drop^S\ m\ xs$ *for* $xs\ ::\ SeqFinSeq\ \alpha,\ m \geq 0$.

Each processor splits its computation into two parts, $work1\ =\ take^S\ m\ xs$ and $work2\ =\ drop^S\ m\ xs$. The parameter m can be calculated so as to minimise the total time; by leaving m as a variable, we are describing a family of related algorithms.

The excess work $work2$ can be offloaded from processors with too much work. In other processors, $work2\ =\ []$. This is done using a communication operation $move^P$ with an inverse $move'^P$ that can be used later to return the partial fold results to the right processor.

Lemma 3. $map^P\ F\ =\ move'^P\ \circ\ (map^P\ F)\ \circ\ move^P$ *for any permutations* $move^P$ *and* $move'^P$ *such that* $move'^P \circ move^P\ =\ id$.

The following program moves the excess work, $work2$, to processors that have room for it. Each processor then computes two fold results, and sends the

second, *res*2, back to its original processor, where it is combined with the first result, *res*1.

$$
\begin{aligned}
maptri\ f\ a\ xss\ =\ &\textbf{let}\ work1\ =\ map^P\ (take^S\ m)\ xss\\
&\quad\ work2\ =\ map^P\ (drop^S\ m)\ xss\\
&\quad\ res1\ =\ map^P\ (foldl^S\ f\ a)\ work1\\
&\quad\ res2\ =\ move'^P (map^P\ (foldl^S\ f\ a)\ (move^P\ work2))\\
&\textbf{in}\\
&zipWith^P\ f\ res1\ res2
\end{aligned}
$$

4 Analysis and Transformation

The next step is to calculate values for m and $move^P$ so as to to minimise the total time. In general, the best performance may not result from an optimal load balance, since the communication time entailed by the load balancing must also be considered. If T_f is low compared to T_{com}, then the time taken to rearrange the data may be higher than the time saved by the improved load balance, so load balancing is not a good idea. On the other hand, if T_f is expensive, then the load balanced program is also the cost optimised one. In general, there are also cases between these two extremes, when moving a few elements can produce a sufficiently good load redistribution to improve the program, but achieving a perfect load balance would require a prohibitively large amount of communication.

There is not space here for a complete formal analysis, but the rest of this section shows the flavour of the calculation. p, the index permutation function of $move^P$, is used to calculate $move^P$, such that $(move^P\ xs)!!i\ =\ xs\ !!\ p\ i$. We assume that $T_f \gg T_{com}$ on the target architecture for all message sizes k, such that $0 \leq k \leq n+1$. Then the total cost is minimised by achieving a perfect load balance, subject to two sub-goals:

1. A processor's work load is proportional to the number of elements it holds. The total number of elements is $\frac{1}{2}n(n+1) \approx n\frac{n}{2}$, so each processor should have about $\frac{n}{2}$ elements.
2. Overloaded processors should only send data, under-loaded processors should only receive, and no processor should do both.

Now the parameters m and p must be calculated. In the original triangular matrix, processor i contains $i+1$ elements. Using this property and standard size lemmas, the number of elements remaining in processor i after load balancing is calculated as follows:

$$
\begin{aligned}
\text{task size } i &= \#((map^P(take^S\ m)xss)!!i) + \#((move^P(map^P(drop^S\ m)xss))!!i)\\
&= \min(m, i+1) + \max(0, p\ i+1-m)
\end{aligned}
$$

The first term in this expression corresponds to data kept and the second term to data received. The expression can be simplified further by considering

the sending and receiving processors separately. Note that the sending processors receive no data, and receiving ones keep all their original data, ie. $i+1$ elements.

The simplified value of processor i's workload can now be equated with the mean workload $\frac{n}{2}$, allowing m and p to be calculated: $m = n`div`2 + 1$ and $p = n - i - 1 \Rightarrow move^P = reverse^P = move'^P$. All that remains is to substitute the chosen values for the variables:

$$
\begin{aligned}
maptri\ f\ a\ xss\ =\ &\textbf{let}\ m\ =\ (length^P\ xss)`div`2\ +\ 1\\
&\quad work1\ =\ map^P\ (take^S\ m)\ xss\\
&\quad work2\ =\ map^P\ (drop^S\ m)\ xss\\
&\quad res1\ =\ map^P\ (foldl^S\ f\ a)\ work1\\
&\quad res2\ =\ reverse^P\ (map^P\ (foldl^S\ f\ a)\ (reverse^P\ work2))\\
&\textbf{in}\\
&zipWith^P\ f\ res1\ res2
\end{aligned}
$$

However, this is not the end of the story. As indicated in the introduction, further transformations, which are not shown in this paper, convert the program into an imperative C+MPI program.

5 Conclusions

In this paper we have demonstrated how static load balancing can be introduced into a data parallel program, using formal program transformations based on a pure functional language. The transformation is introduced cleanly, without having to worry about other details involved in parallel programming. The programming methodology uses a library of suitable distributed types, parallel operations, collective communication operations, and lemmas.

Topics for future research include methods for introducing other optimisations, extensions to the library of combinators and their lemmas, and tools providing partially automated support for the programming process.

References

1. Joy Goodman. A methodology for the derivation of parallel programs. Workshop UMDITR03, Departamento de Informática, Universidade do Minho, September 1998. 535
2. Sergei Gorlatch. Stages and transformations in parallel programming. In *Abstract Machine Models for Parallel and Distributed Computing*, pages 147–162. IOS Press, 1996. 535
3. Kevin Hammond and Greg Michaelson, editors. *Research Directions in Parallel Functional Programming*. Springer, 1999. 535
4. Thomas Rauber and Gudula Rünger. The compiler TwoL for the design of parallel implementations. In *Proceedings of the 4th International Conference on Parallel Architecture and Compilation Techniques*, pages 292–301. IEEE Computer Society Press, 1996. 535