The Skel-BSP Global Optimizer: Enhancing Performance Portability in Parallel Programming

Andrea Zavanella

Dipartimento di Informatica
Università di Pisa - Italy
zavanell@di.unipi.it
http://www.di.unipi.it/∼zavanell

Abstract. The paper describes the Skel-BSP Global Optimizer (GO), a compile-time technique tuning the structure of skeletal programs to the characteristics of the target architecture. The GO uses a set of optimization rules predicting the costs of each skeleton. The optimization rules refer to a set of implementation templates developed on top of the EdD-BSP (a variant of the BSP model). The paper describes the Program Annotated Tree representation and the set of transformation rules utilized by the GO to modify the starting program. The optimization phases: balancing, scaling and augmenting are presented and explained running the GO on a cluster of PCs for an image analysis toy-program.
key words: skeletons, BSP, optimization, performance portability.[1]

1 Introduction

The Skel-BSP [14] approach has been proposed to conjugate Skeletons [3, 5] and BSP [11] to obtain high level programming and performance portability. The paper presents the Global Optimizer (GO) a compile-time technique tuning Skel-BSP programs to the target architecture. GO uses a set of transformation rules preserving the program semantics and chooses the distribution of processors among the program components. The paper describes the "global" approach utilized by Skel-BSP on top of the "local" optimizations embedded in each implementation template [15, 12, 13]. These rules are based on a BSP-like computational model: the EdD-BSP (see Section 2.2). The paper shows how these two strategies work together to optimize the EdD-BSP intermediate code on a given parallel platform. The GO is presented by describing its main procedures: `augmenting, balancing` and `scaling`. An example of the GO behavior is provided compiling an image analysis toy-program on a cluster of PCs (Backus).

[1] This work has been supported by the Italian M.U.R.S.T. within the Mosaico framework

A. Bode et al. (Eds.): Euro-Par 2000, LNCS 1900, pp. 658–667, 2000.

2 The Skel-BSP Methodology

2.1 The Skel-BSP Compiler

Skel-BSP forces the programmer to concentrate on exposing a "parallelization strategy" more than a parallel algorithm. Therefore the programmer expertise is exploited in the direction of writing a composition of already defined parallel patterns (`Pipe,Farm,Map,Reduce`) according to the P3L programming style [4, 10]. The Skel-BSP compiler derives an optimized implementation using three additional sources:

- a set of BSP-lib [8] implementation templates ,
- a set of performance equations [14],
- the EdD-BSP parameters of the target architecture,

The "local" optimizations are stored in a set of reusable components (templates) with associated two optimization rules expressed by the following equations:

- $T_{opt}(param, M)$: the optimal service time on a given EdD-BSP computer M (see Section 2.2);
- $N_{opt}(param, M)$: the minimal number of workers obtaining the optimal service time on M;

The tuple of application dependent parameters ($param$) is computed using a sequential profiling while the EdD-BSP parameters one (M) is provided by the parallel profiling phase (see Fig. 1).

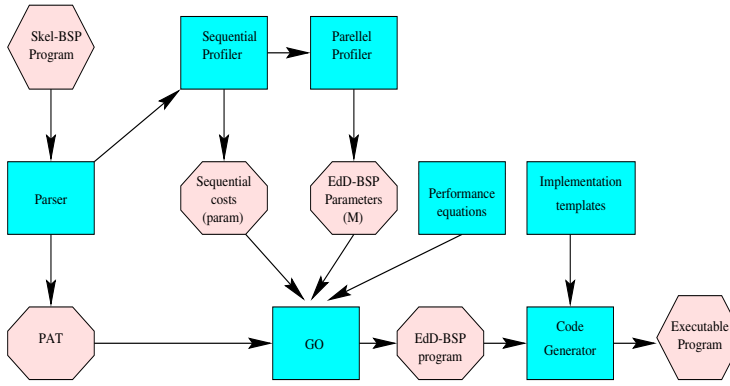

Fig. 1. The structure of the Skel-BSP compiler

2.2 The Cost Model

The EdD(Edinburgh-Decomposable)-BSP model , has been introduced as a variant of the BSP [11] to predict skeletal programs performance. A BSP computer is a set of p couples processor-memory interconnected in order to be able to communicate point-to-point and to perform a global synchronization. A BSP computation is organized as a sequence of synchronous supersteps including (a) a local computation phase, (b) a global communication phase and (c) a barrier synchronization. The cost of each superstep is given by: $T_{sstep} = W + hg + L$ where W is the maximum amount of work performed in the local computation phase, and h is the maximum number of messages sent or received during the communication phase. The parameters g and L are the "standard" BSP parameters defined as the costs to send a single message (g) and to perform a barrier synchronization (L). The EdD-BSP variant introduces two extensions of the BSP model: a couple of parameters g_∞ and $N_{1/2}$ in place of g modeling the communication bandwidth as a function of the message size (see [9], and the decomposability (see the work of Kruskal et al.[6]). An EdD-BSP computer is then a tuple parallel M including four parameters:

$$M = (l, g_\infty, N_{1/2}, p)$$

A relevant innovation introduced by the second extension is the possibility of `partitioning` a BSP computer in submachine. Each submachine acts as an autonomous BSP computer (i.e. it synchronizes independently). The model admits two kinds of supersteps: the computational supersteps, and join/partition supersteps which costs are stated in the following equations:

$$T_{sstep} = \begin{cases} W + hg_\infty(\frac{N_{1/2}}{h} + 1) + L & \text{computational step} \\ L & \text{join-partition} \end{cases}$$

Assuning that at a given time the p processors are partitioned in $q < p$ submachines, the cost to perform a superstep is expressed as the maximum cost for each submachine to reach the next join operation. This means that T_{sstep} has to be computed recursively as the maximum time to execute the EdD-BSP program running on the i machine. Assuming that no other partition is executed we would obtain:

$$T_i = \sum_{j=1}^{nstep(i)} T_{sstep}(i, j)$$

Where $nstep(i)$ is the number of supersteps performed by the submachine i and $T_{sstep}(i, j)$ is the "classic" BSP cost for the j-th superstep of the i-th submachine. This extension enables EdD-BSP to predict the execution costs of skeletal programs whose components require different number of synchronizations. A practical implementation of a decomposable BSP programming has been realized in the Paderborn University BSP library (PUB) [2]. The need for the EdD-BSP model and the results of predicting the cost of skeleton programs using such a model are included in [15, 12, 13].

3 The Program Annotated Tree (PAT)

The Program Annotated Tree is an extension of the syntax-tree of a Skel-BSP program, where each node includes three fields: $(Skel(param), N_w, T_{serv})$. The $Skel(param)$ field contains a skeleton identifier ($Skel$) and the list ($param$) of performance parameters (i.e. in Fig. 2 the sizes of input output structures d_0, d_1, d_2). The field N_w contains the number of processors used by the module and T_{serv} is the service time of the subprogram rooted at the node. An example of a node of the PAT is shown in Fig. 2. The initial values stored in the PAT $param$ fields are computed by the sequential profiling phases while the other fields are filled by the GO during the `init` phase.

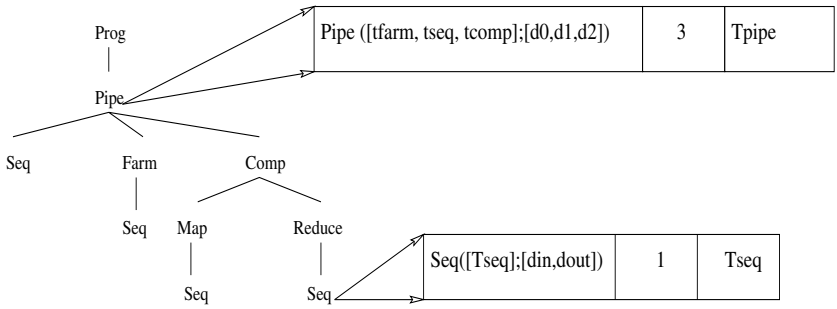

Fig. 2. The PAT of a Skel-BSP program

4 The Global Optimizer

4.1 The Transformation Rules

The GO transforms valid Skel-BSP programs (see the grammar in [14]) using the transformation rules in Tab. 1 to adapt the program to the characteristics of the target machine. The rules 3 and 4 refer to the `Comp` constructor which models the sequential composition of Data Parallel modules. The rule 5 uses the `concat` operator which makes a monolitique sequential constructor from a sequence $X_1, \ldots X_k$ of sequential modules. Fig. 3 shows an example of two valid transformations preserving the semantics of the starting program. The two programs result from two different sequences of program transformations: `(b)` is obtained from `(a)` using 1-3-6, while `(c)` is obtained using 2-7-5.

4.2 Initializing the PAT

The overall structure of GO is shown in Fig. 4. The `init` procedure computes the values of N_w and $Tserv$ for the leaves of the PAT then propagates the results up

Num.	Rule	Name
1	$Seq \longrightarrow Farm(Seq)$	Farm insertion
2	$Farm(Seq) \longrightarrow Seq$	Farm elimination
3	$Comp(X_1, \ldots X_k) \longrightarrow Pipe(X_1, \ldots X_k)$	Pipe insertion
4	$Pipe(X_1, \ldots X_k) \longrightarrow Comp(X_1, \ldots X_k)$	Pipe elimination
5	$Pipe(X_1, \ldots X_k) \longrightarrow concat(X_1, \ldots X_k)$	Pipe collapse
6	$Pipe(X_1, Pipe(Y_1, \ldots Y_h), \ldots X_k) \longrightarrow Pipe(X_1, Y_1, \ldots Y_h, \ldots X_k)$	Pipe fusion
7	$Pipe(X_1, Y_1, \ldots Y_h, \ldots X_k) \longrightarrow Pipe(X_1, Pipe(Y_1, \ldots Y_h), \ldots X_k)$	Pipe distribution

Table 1. The GO transformation rules

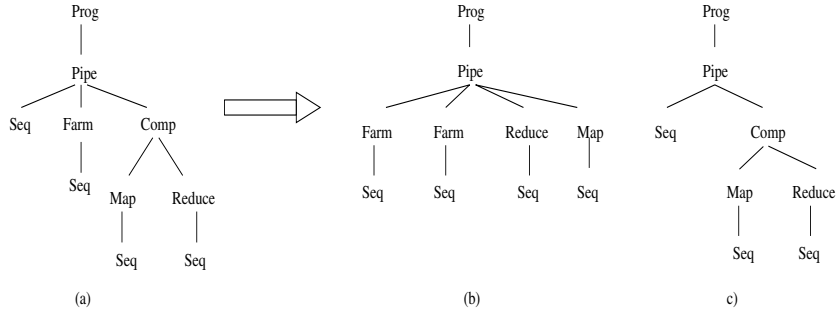

Fig. 3. Two transformations of a valid program

to the root. In this phase the optimization rules for each skeleton are computed on the target M_∞ which allows to saturate each parallel components with the maximum useful parallellism: $M_\infty = (l, g_\infty, N_{1/2}, \infty)$. The PAT for M_∞ is called *fully parallel version* of the program. The pseudo-code of the `init` procedure is given in Fig. 5. The first loop computes the values of T_{serv} and N_w for: Farm, Map, Reduce and Scan. The functions `Topt(Skel,`M_∞`)` and `Nopt(Skel,`M_∞`)` return the optimal values according to the optimization rules defined in [12, 15]. The second loop propagates the values of T_{serv} and N_w to the higher layers of the tree. We have three optimization cases:

1. $N_w > p$: GO reduces the number of processors minimizing the loss of performance;
2. $N_w \leq p$: GO improves the program performance by adding processors;
3. $N_w = p$: GO terminates;

4.3 Reducing Resources

The goal of this phase is to reduce the number of processors to match the number of available processors while minimizing the loss of performance. The reduction takes place in two subphases:

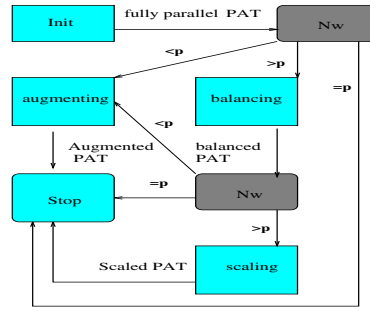

Fig. 4. The GO structure

```
For all Node in PAT such that:
       ((Node.Skel=Farm or
         Node.Skel=Map or
         Node.Skel=Reduce or
         Node.Skel=Scan or
     Node.Skel=Comp) and
         marked(Node)=false)
       Node.Nw=Nopt(Node.skel,M)
       Node.Tserv=Topt(Node.skel,M)
       mark(Node))
endfor (phase 1)
```

```
For all Node in PAT such that:
  ((Node.Skel=Pipe or
        Node.Skel=Comp) and
        marked(children(Node)) and
        marked(Node)=false)
  Update(Node.Skel)
      Node.Nw=Nopt(Node.skel,M)
      Node.Tserv=Topt(Node.skel,M)
      mark(Node)}
endfor
(phase 2)
```

Fig. 5. The two phases of the GO init algorithm

1. *balancing*: in this phase the number of processors employed by each pipeline stage with service time $T_{serv} < T_{slow}$ (where T_{slow} is the service time of the slowest stage) is "reduced".
2. *scaling*: when the balancing phase is completed, if N_w is still larger than p, the program must be scaled.

The `balance` procedure computes the minimum number of processors mp such that: $Tserv \leq T_{slow}$. In the case: $mp = 1$ the suitable elimination rule is applied and the PAT is transformed by setting: $Node.skel = Seq$. At the end of the balancing phase, three optimization cases may occur:

1. $N_w = p$ GO terminates;
2. $N_w < p$ GO performs the augmenting phase;
3. $N_w > p$ GO performs the scaling phase;

The difficulty of scaling the program arises from the fact that many transformations may reduce the number of processors and GO must select which one leads to the optimum. Other works have demonstrated [1] that using a gradient method the optimizer may stop in a local optimum while the global optimum

could have been reached by accepting not optimal moves along the computation. Therefore the choice made in GO is to apply an exhaustive search. GO generates the transformations that reduce the value of N_w to p creating a sequence of sets $scal\_PAT(i)$ containing the PATs obtained using $N_w - i$ processors. The solution must be located in $scal\_PAT(N_w - p)$ where all the PATs are filled and a simple search of the minimum service time is performed to find the best implementation. We have the following assertion:

Assertion 1 (scaling) *The complexity T_{scal} of scaling is bounded by:*

$$T_{scal} \leq (N_{tr} \star N_{sk})^{N_w - p} \star N_{sk}$$

Where N_{tr} is the number of transformations of our system and N_{sk} is the number of skeletons in the program.

Since the transformations reducing the number of processors are: (a) cutting processors from `Farm, Map etc.`, (b) replacing a `Pipe` with `Comp`, (c) collapsing a `Pipe` to a `Seq`, in our system $N_{tr} = 3$.

4.4 Augmenting Parallelism

This phase exploits the available $p - N_w$ processors to minimize the program cost. Since the starting PAT uses the maximum parallelism for the user program, the program must be transformed in order to increase parallelism. Two types of suitable transformations can be applied: (a) the insertion of `Pipe` in place of `Comp` (b) the `Farm` insertion (since the other parallel constructors already use the optimal number of processors). We have the following assertion:

Assertion 2 (Augmenting) *the complexity of augmenting is bounded by:*

$$T_{aug} < (N_{Seq} + N_{Comp})^{(p - N_w)} \star N_{skel}$$

where N_{Comp} and N_{Seq} are the number of `Comp` *and* `Seq` *constructors respectively.*

The cost of the augmenting algorithm is furtherly reduced using a pruning technique to decrease the number of generated solutions. In practice the only PATs generated and compared are those in which the number of processors is smaller than p. The pseudo code of the augmenting algorithm is shown in Fig. 6. The procedure `Enumerate` inserts in `solutions` the allocations of processors to constructors satisfying the constraint that the number of processors assigned does not exceed N_{opt}. The Procedure `Prune` eliminates the solutions where the number of processors exceeds p. Finally the procedure `Generate` builds the suitable PATs and `Findopt` selects the solution with the minimum $Tserv$.

5 Case Study

A simple example of the behavior of the GO procedures has been provided using as a case study an image analysis toy-program: IA. IA is a synthetic program

```
int assigned[1..Nseq+Ncomp];
int max[1..Nseq+Ncomp];
int solutions[1..maxsol;1..Nseq+Ncomp];
int Ntr,Nsol;
real Tserv[1..maxsol];
Ntr=Nseq+Ncomp;
for i = 1 to Ntr
    select(Node,i);
    max[i]=max(Nopt(Node.skel,M),p-Nw)
endfor
Enumerate(assigned,max,solutions);
Prune(Solutions,Nsol);
Generate(Solutions,Tserv);
Findopt(Solutions,Tserv,assigned);
```

Fig. 6. The GO augmenting algorithm

including a four stages pipeline where the stages implement: (1) the input from file; (2) the filtering of data; (3) a convolution like computation; (4) the output on file. The sintax tree of IA written in Skel-BSP is shown in Figure 7. The application has been profiled on top of Backus a 10 PCs cluster with PII 266 Mhz CPU and 128 Mbytes RAM running Linux. The machine parameters (M) and the IA *param* list are shown in Tab. 2. Using the values in Tab. 3 the GO produces the transformation visualized in Fig. 8.

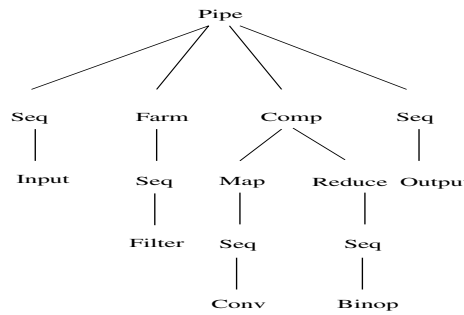

Fig. 7. The Skel-BSP structure of IA

6 Conclusions and Related Work

The paper shows that GO may adapt the structure of a skeletal application to optimize its running time on a specific target architecture. Skel-BSP programs

	g_∞ (μ sec)	L (μsec)	$N_{1/2}$ (bytes)	p (processors)	X
M (Backus)	0.8	1500	64	10	X
	t_{in}	t_{filter}	t_{conv}	t_{red}	t_{out}
T_{seq}(msec)	10	20	80	30	15
	d_{in}	d_{filt}	d_{conv}	d_{red}	d_{out}
sizes (Mbytes)	32	32	32	32	32

Table 2. The parameters of Backus and the IA param list

Skeleton	N_{opt} (proc)	T_{opt} (msec)
Farm	6	8
Map	10	10
Reduce	10	4

Table 3. The values of Nopt and Topt for the IA example on Backus

must be simply recompiled to modify their parallel behavior, therefore Skel-BSP seems a promising approach to reach performance portability. The GO exploit a set of formula providing the costs of optimized implementation of each skeletons based on the EdD-BSP cost model. A related approach is the transformational frame proposed by Gorlatch et al. [7] where an "ad hoc" cost model is proposed to drive a semi-automatic transformation system. The future development of the GO will include:

- evaluating a general approch to efficiently implement the current GO optimization algorithm;
- an extensive validation on several parallel architectures;

finally the set of transformation rules will be enlarged.

References

[1] M. Aldinucci, M. Coppola, and M. Danalutto. Rewriting skeleton programs: how to evaluate the data-parallel stream-parallel tradeoff. In *Proceedings of International Workshop on Constructive Methods for Parallel Programming*, number MIP-9805 in Technical Report, University of Passau, 1998.

[2] O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The paderborn university bsp (pub) library - design, implementation and performance. In *Proceeding of 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP)*, 1999.

[3] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* The MIT Press, Cambridge, Massachusetts, 1989.

[4] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. In D. B. Skillicorn and D. Talia, editors, *Programming Languages for Parallel Processing.* IEEE Computer Society Press, 1994.

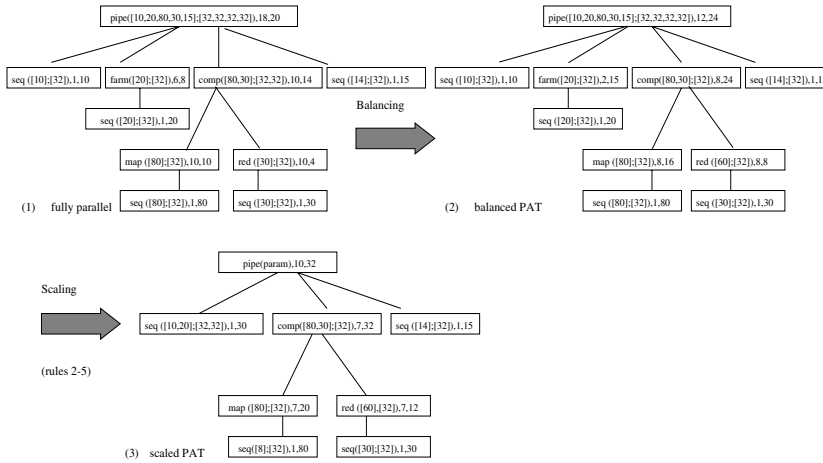

Fig. 8. The optimization of IA

[5] J. Darlington, Y.-K. Guo, Hing Wing To, and J. Yang. Functional skeletons for parallel coordination. *Lecture Notes in Computer Science*, 966:55–67, 1995.

[6] P. De La Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. *Lecture Notes in Computer Science*, 1124:352–360, 1996.

[7] S. Gorlatch and S. Pelagatti. A transformational framework for skeletal programs: Overview and case study. In Jose Rohlim, editor, *Proc. of Parallel and Distributed Processing. Workshops held in Conjunction with IPPS/SPDP'99*, volume 1586 of *LNCS*, pages 123–137, Berlin, 1999. Springer.

[8] J. M. D. Hill, B. McColl, D. — C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. H. Bisseling. BSPlib: The BSP programming library. *Parallel Computing*, 24(14), 1998.

[9] R. Hockney. Performance parameters and benchmarking of supercomputers. *Parallel Computing*, 17(10-11):1111–1130, December 1991.

[10] S. Pelagatti. *Structured Development of Parallel Programs.* Taylor & Francis, 1997.

[11] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[12] A. Zavanella. Optimising Skeletal Stream Parallelism on a BSP Computer. In P. Amestoy, P. Berger, M. Dayde, I. Duff, V. Fraysse, L. Giraud, and D. Ruiz, editors, *Proceedings of EURO-PAR'99*, number 1685 in LNCS, pages 853–857. Springer-Verlag, 1999.

[13] A. Zavanella. Skel-Bsp: Performance Portability for Skeletal Programming. In M. Bubak, H. Afsarmanesh, R. Williams, and B. Hertzberger, editors, *Proceedings of HPCN 2000*, volume 1823 of *LNCS*, pages 290–299, 2000.

[14] A. Zavanella. *Skeletons and BSP: Performance Portability for Parallel Programming.* PhD thesis, Dipartimento di Informatica Univesita di Pisa, 2000.

[15] A. Zavanella and S. Pelagatti. Using BSP to Optimize Data-Distribution in Skeleton Programs. In P. Sloot, M. Bubak, A. Hoekstra, and B. Hertzberger, editors, *Proceedings of HPCN99*, volume 1593 of *LNCS*, pages 613–622, 1999.