How Portable is Nested Data Parallelism?

Manuel M. T. Chakravarty[1] and Gabriele Keller[2]

[1] Inst. of Inform. Sciences and Electronics, University of Tsukuba, Japan
chak@is.tsukuba.ac.jp; www.score.is.tsukuba.ac.jp/~chak/
[2] School of Computing Sciences, University of Technology, Sydney, Australia
keller@socs.uts.edu.au; www.socs.uts.edu.au/~keller/

Abstract. Research on the high-performance implementation of nested data parallelism has, over time, covered a wide range of architectures. Scalar and vector processors as well as shared-memory and distributed memory machines were targeted. We are currently investigating methods to integrate this technology into a single portable compiler back-end. Essential to our approach are two program transformations, flattening and calculational fusion, which even out irregular parallelism and increase locality of reference, respectively. We generate C code that makes use of a portable, light-weight, collective-communication library. First experiments on scalar, vector, and distributed-memory machines support the feasibility of the approach.

1 Introduction

We discuss the design and code quality of a compilation system that implements nested data parallelism on a wide range of parallel machines; we strive for portable parallelism over machines containing scalar or vector processors and using shared or distributed memory. In a feasibility study, we are currently evaluating our approach on different platforms—as different as the Cray T3E and Fujitsu's VX line of vector processors. We achieve portability by combining the high-level parallel programming model of nested data parallelism with a rich program transformation system that can be parametrised with details of the target architecture, and generate code based on a portable collective-communication library based on one-sided communication.

Nested data parallelism is a flexible programming model that allows to code a wide range of parallel algorithms and comes with a language-based cost model [5]; compared to flat data parallelism, it simplifies coding irregular computations and data structures, such as sparse matrices. However, the machine independence and convenience for the programmer come at the cost of an increased challenge for the compiler writer.

Blelloch & Sabot [7, 3] introduced the so-called *flattening transformation*, which converts a nested parallel to an equivalent flat program without reducing the parallelism specified in the original program. Despite the importance of this transformation, on its own, it is not sufficient for producing good code for modern processors. Subsequent work—in particular, Chatterjee's thesis [9] and

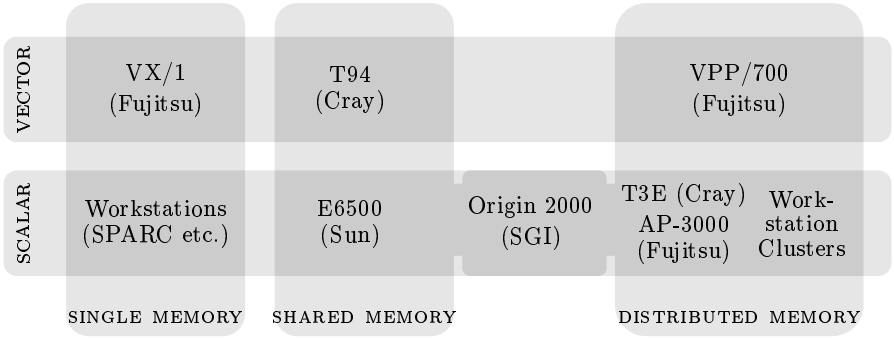


Fig. 1. Architecture space

the second author's thesis [14]—demonstrated that flattened code can indeed be successfully compiled to modern machines. In the approach discussed in this paper, we optimise flattened code using *calculational fusion* [19, 14] and generate C code that uses collective-communication operations, which are implemented using one-sided communication; the latter allows a uniform view on different memory models.

In summary, the paper makes the following three contributions: (1) It proposes a design for a highly portable implementation of nested data parallelism; (2) thereby, it briefly introduces the design of a portable library of collective operations that implements the runtime system of the generated code; and (3) it evaluates the proposed design by a set of benchmarks on a range of different architectures. The discussed benchmarks are not extensive enough to allow a complete evaluation of the system, but they substantiate the feasibility of the approach and provide a guideline for further optimisations.

The paper is structured as follows. Section 2 reviews the architectures that we target. Section 3 briefly introduces nested data parallelism and gives an overview over our compilation system. Section 4 discusses the impact that program transformations and the collection-oriented communication library have on portability. Section 5 summarises and evaluates the benchmarks. Finally, Section 6 briefly reviews related work and concludes.

2 The Architecture Space

Previous work addressed the implementation of nested data parallelism on a range of different machines, such as vector processors [3, 10], shared-memory multiprocessors [9, 10], and distributed-memory machines [6, 14]. However, the implementations, while being based on flattening, used different optimisation techniques and enjoyed various levels of success. We are investigating to which extend we can target the full range of architectures with a uniform compilation system controlled by a set of parameters characterising the target machine and still obtain efficient code. Figure 1 displays the targeted architecture space, where we categorise high-performance architectures by processor type and

memory system. It should be noted that the listed systems require parallelism to be expressed in different forms to achieve maximal performance. For example, vector processors perform best on regular computations where long vectors are repeatedly combined using elementary operations, such as addition, multiplication, and so on. On the other hand, modern RISC processors favour complex operations on a small set of scalars, as this leads to better register and cache utilisation. It is thus challenging to devise a single compilation technique that allows to generate good code for both architectures, especially under the added burden of dealing with either a shared- or distributed-memory model.

We argue in this paper that nested data parallelism can indeed be efficiently mapped to these architectures by a compilation technique that combines two program transformations with a portable communication library based on one-sided communication. The two program transformations are *flattening* [7, 3] and *calculational fusion* [19, 14], and they are central in tuning the code for varying processor architectures. Flattening transforms a nested data parallel program into a flat data parallel program operating on long vectors. This makes the code better suited for vector processors and it is also an essential part of our load balancing strategy. Calculational fusion eliminates unnecessary intermediate structures, makes the code cache friendly, and allows to optimise communication patterns; the degree of fusion and the exact fusion rules depend on the architecture. Keller [14] showed how to combine these two transformations for the high-performance implementation of nested data parallelism. The communication library has efficient implementations on shared- and distributed-memory systems as the underlying model of one-sided communication provides an efficient and convenient abstraction of both shared memory and distributed memory (in our experience, it does so better than conventional message passing).

3 Nested Data Parallelism and the Nepal System

Although we concentrate on the use of nested data parallelism in functional languages, the techniques are also applicable to imperative languages [8, 1, 10]. In the following, we briefly introduce the programming model and the structure of our compilation system.

3.1 The Programming Model

The advantages of nested data parallelism as a parallel programming model are discussed in detail elsewhere [5]; hence, we restrict ourselves to a brief introduction. Our examples use the language *Nesl* [4], which expresses parallelism by builtin parallel operations, such as reductions and scans, and an *apply-to-each* construct. The latter evaluates some expression—called the *body*—for each element of one or more vectors; like in $\{x * y \ : \ x \ \mathbf{in} \ xs, \ y \ \mathbf{in} \ ys\}$, where the elements of xs and ys are multiplied pairwise. By applying builtin parallel operations or using other *apply-to-each* constructs in the body, we express *nested* parallelism;

like in $\{add\_reduce\ (xs)\ :\ xs\ \mathbf{in}\ xss\}$, which computes the sum for each subvector xs of xss—e.g., for [[1, 2, 3], [], [4, 5]], we have [6, 0, 9].

Nesl is a typed language: A type $[\alpha]$ contains vectors (or lists) with element type α. A nested vector $[[\alpha]]$ is composed from subvectors of type $[\alpha]$, which can be of arbitrary length, i.e., such a structure may be arbitrarily irregular. A type $(\alpha_1, \ldots, \alpha_n)$ contains n-tuples with components of type α_i. Nested data parallelism was originally restricted to computations where parallelism is solely expressed over nested vectors, but we showed recently how to extend it to tree-based parallel code, which, for example, is useful to implement hierarchical tree codes solving the n-body problem [15].

3.2 Sparse-matrix Vector Multiplication

As an example, let us consider sparse-matrix vector multiplication where the matrix is in compressed row format. More precisely, we represent dense vectors by values of type $[Float]$, sparse vectors by index-value pairs of type $[(Int,\ Float)]$, and sparse matrices by vectors of sparse vectors, i.e., values of type $[[(Int,\ Float)]]$. For example, the matrix

$$\begin{bmatrix} 5 & 0 & 8 \\ 0 & 0 & 0 \\ 9 & 0 & 0 \end{bmatrix} \text{ is represented by } [[(0,5),(2,8)],[],[(0,9)]].$$

Given this representation, the multiplication of a sparse matrix with a dense vector is implemented by the following function $smvm$,

$$\begin{array}{l}\mathbf{function}\ smvm\ :\ (mat:\ [[(Int,\ Float)]],\ vec:\ [Float])\ \rightarrow\ [Float]\ = \\ \quad \{add\_reduce\ (\{vec[i] * x\ :\ (i,\ x)\ \mathbf{in}\ row\})\ :\ row\ \mathbf{in}\ mat\};\end{array}$$

For each row of the matrix mat, the function multiplies the non-zero elements with the corresponding vector elements and sums up all products within a row. It should be clear that this computation is depending on the matrix structure and, in general, is highly irregular.

3.3 The Compilation System

To verify the feasibility of our approach, we are currently implementing a compiler that will eventually target the whole range of machines outlined in Section 2. The structure of this compiler is displayed in Figure 2. Outside of the grey area are the compiler phases that are commonly employed in flattening-based implementations of nested data parallelism. First, we convert the nested data parallel language into a *nested kernel language*, i.e., the input is type checked and all syntactic sugar removed. Then, we apply the flattening transformation to convert all nested in flat parallelism.[1] Inside the grey area are the phases that

[1] The interested reader finds the language definition of the nested and the flat kernel language on our project page [17]. Together with code of the system as far as it is implemented.

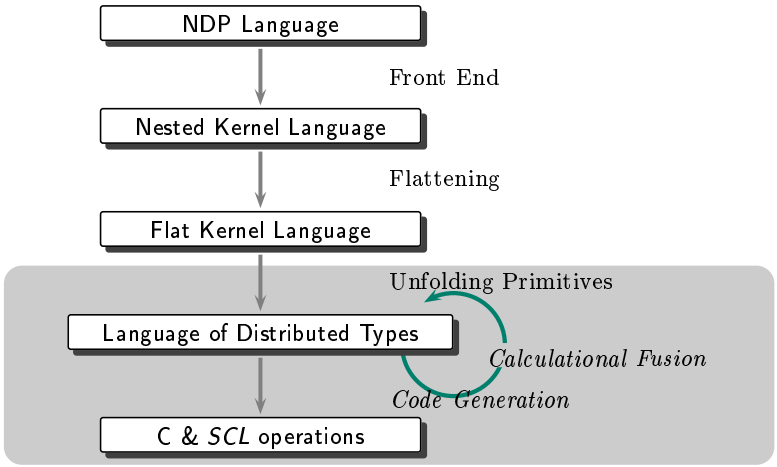


Fig. 2. The Nepal Compilation System

are new in our approach. As discussed in detail in [14, 16], *unfolding of primitives* decomposes the data parallel primitives into their purely local and their global components, using an intermediate language that distinguishes between local and global values by the type system. In this representation, we apply *calculational fusion* to optimise local computations and communication operations for the target architecture. Finally, the code generation phase produces C code that uses our collective-communication library SCL to maintain distributed data structures and to specify communication. The library internally maps all collective communication to a small set of one-sided communication operations.

As mentioned in the previous section, it is essentially the combination of flattening with calculational fusion and the communication library that allows us to target a wide range of high-performance architectures. Furthermore, the components that are marked by use of an *italic* font in Figure 2 behave differently in dependence on the targeted architecture—we call them *target-dependent components*. However, the flattening transformation, while being essential for our approach to portability, operates in the same way for all kinds of target architectures; it does not specialise the code for an architecture, but generally brings it into a form that makes it easier for subsequent phases to generate good code. In contrast, application of calculational fusion, code generation, and the SCL library have to be parametrised with information about the target architecture to generate good code. The next section discusses this in more detail.

4 Program Transformation and the Communication Library

Let us consider the properties of the machine architectures of Figure 1 (of Section 2) in combination with the stages of the Nepal compilation system from

Figure 2 (of Section 3). Roughly speaking, we have to take care of the following properties:

- Vector processors need *regular* computations (to keep the vector pipeline filled) and *uniform memory access* (to allow vectorisation). On the other hand, scalar processors need *localised memory access* (for cache friendliness). These goals are partially conflicting.
- Multiprocessors have to *minimise synchronisation*.
- Shared memory requires *direct memory access* (for high memory bandwidth). Furthermore, distributed memory requires *even data distribution* (to avoid excessive communication) and a good tradeoff *between load balancing and data re-distribution*.

The compiler achieves these properties as follows:

- Flattening separates data from shape[2] computations: This leads to regular computations and even data distribution.
- Calculational fusion transforms fine-grained vector loops into deep computations: This localises memory access, reduces synchronisation, and allows to trade load balance for data re-distribution.
- Our code generator uses varying templates for reductions, scans, etc.: This allows to resolve the conflicting goals for vector and scalar processors.
- The SCL library is based on one-sided communication: This minimises synchronisation and allows direct memory access.

The combined effect of flattening and calculational fusion is depicted in Figure 3. The top part (a) of the figure visualises the structure of *smvm* without the extraction of the vector elements by $vec[i]$, i.e., the dark vector represents the non-zero elements of the matrix *mat* and the light vector the corresponding elements taken from the vector *vec*. The partitioning of the two vectors represents subvectors, which contain the elements of a single row of the matrix each. The program multiplies the vectors elementwise, and then, sums up all those values that are in the same subvector. The dots represent the individual elements of the result vector. The partitioning depends on the structure of the matrix and is, in general, irregular. Thus, it is hard to execute the computation efficiently on any of the mentioned architectures, apart from a single scalar processor.

The flattening transformation converts the original program into a flat data parallel program whose behaviour is depicted by part (b). The value and shape information of the multiplied vectors are now separated. Multiplication executes without any reference to the shape information, which is represented as a *segment descriptor*. The summation of the products is implemented by a segmented reduction [11], which computes all sums in a single parallel operation. This code is useful for a single vector processor (although it can still be optimised); however, it is unsuitable for modern scalar processors, as it exhibits poor

[2] We understand *shape* as the structure information remaining after all primitive data like numerals are removed from a data structure—see [13].

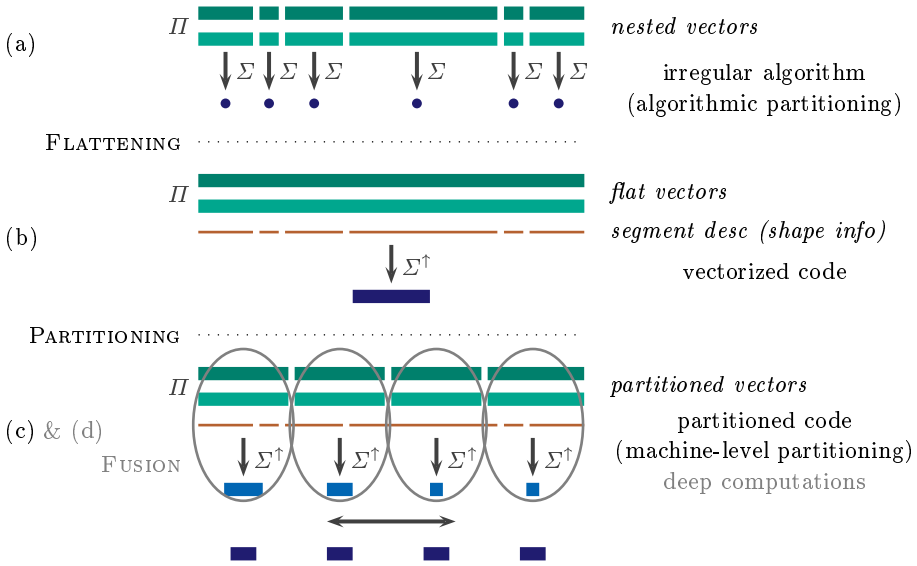


Fig. 3. Effect of the program transformation phases on *smvm* (without the subexpression $vec[i]$)

cache behaviour. Furthermore, partitioning over multiple processing elements is still implicit.

In the first step of the transformation of the flattened code, the computation is partitioned into independent subcomputations, resulting in the computation depicted in (c)—i.e., the lower most part (the grey ellipsoids should be ignored here, as they sketch the effect of the next step). During partitioning (which happens as part of the unfolding of primitives from Figure 2), vectors are uniformly split into as many chunks as we have processing elements.[3] The most interesting change in this stage is a consequence of the mismatch between the initial algorithmic partitioning, determined by the segment descriptor and the current physical partitioning. A single segment of the input vectors can be split across different partitions of the physical partitioning, which requires a post-processing step correcting the final sums. More details about the partitioning and its implementation by program transformation are available in [14, 16].

Finally, calculational fusion transforms the program code, for part (d) (including the ellipsoids), such that within a single partition computations are performed vertically as long as no synchronisation between the partitions is necessary. This transformation brings the code into a cache friendly form, but is

[3] The compiler does not fix the concrete number of processors, but introduces as constant P, which is used to partition all data structures and loops. The actual value for P is determined by the runtime system during startup.

only used in a restricted way for vector processors, as we want to preserve the horizontal processing style for them. Details on the specific fusion transformation used in this system are also available in [14, 16]. In the following, we have a closer look at the target-dependent components of the Nepal system.

4.1 Calculation Fusion

The use of calculational fusion for optimising sequential traversals of data structures has received considerable attention, as it can reduce the number of traversals and intermediate data-structures. It is obvious that flattening-based implementations can profit from calculational fusion, as flattening leads to excessive data-structure traversals. However, the techniques for sequential programs cannot be used directly on parallel programs as we have to distinguish between subcomputations that can be executed independently on each processor and global computations that require communication. The following small example, where foo_2 is the fused version of foo_1 demonstrates the problem ($\#xs$ denotes the length of the vector xs, $head(xs)$ the first element of xs, and $tail(xs)$ all but the first element).

$$
\begin{aligned}
foo_1\ (xs,\ ys) &= add\_reduce\ (\{x\ *\ y\ :\ x\ \mathbf{in}\ xs,\ y\ \mathbf{in}\ ys\}) \\
foo_2\ (xs,\ ys) &= \mathbf{if}\ (\#xs\ ==\ 0)\ \mathbf{then}\ [] \\
&\quad\ \mathbf{else}\ (head\ (xs)\ *\ head\ (ys)\ +\ foo_2\ (tail(xs),\ tail(ys))
\end{aligned}
$$

Both specify the same computation, namely the elementwise multiplication and subsequent summation of two vectors. The two vector traversals in foo_1 are combined into a single traversal in foo_2, and the computation produces no intermediate result. However, the parallel implementation of the original computation is straight forward, whereas the parallel implementation of the second variant is far from clear. The reason why the parallel structure becomes obfuscated is quite obvious in this example: By fusing add_reduce, which needs the cooperation of all processors, into a single recursive traversal, we hide the dependencies. Thus, before fusing the parallel computations, we refine the definition and make global and local computations explicit.

The "Language of Distributed Types" (DKL) of Figure 2 supports this by providing values of local and global types as well as language constructs to identify purely local operations. Nevertheless, the exact decomposition of a particular operation into local and global operations is not fixed, and the optimal solution often depends on the target architecture. Thus, we treat the refined definitions as input into the transformation system that allows architecture dependent optimisations on an abstract level. Then, after we have done the decomposition, we can treat the local computations just like sequential operations and fuse them into more complex operations. Furthermore, we can combine subsequent communication operations in some cases.

For fusion, the transformation has to be able to look into the definition of the functions. This can be done in two ways: (a) [20] uses simply the recursive definition of the functions provided by the user or the library definition; and

(b) [18] and [12] use special representations that simplify the transformation. Those representations either have to be provided by the programmer or are automatically deduced by a transformation system. In our case, the decision is easy; we only need the definition of a fixed set of operations, and we provide them in fusion friendly form. However, we do not reuse the representations of [18] or [12], as they operate on recursively defined data-structures and not on indexed structures; in our case, a loop-based representation is more adequate. The loop representation of local computations in our intermediate language is actually less general than both [18] and [12], but this comes as an advantage: It still suffices for our purposes, and it simplifies the transformation considerably.

This subsection give only a rough idea about the way fusion is integrated into the compilation system. A more detailed description and technical discussion can be found in [14]

4.2 Code Generation

The code generation phase has to handle two types of kernel language operations: global and local operations. Global operations are trivial, as they map to corresponding library functions. This leaves only the local operations to be dealt with. On most architectures this means that the code generation does not explicitly have to consider parallelism. However, as will be described shortly, this is not completely true if vector processors are involved. On a conventional processor, each loop-construct of the kernel language is translated into a corresponding C loop. The program transformation already fused successive loops, and it grouped together loops that traverse over structures of same size. Segmented loops constructs are implemented by nested C loops.

In contrast to conventional processors, for which the generation of the loop code poses no substantial problems, the code generation for vector processors is tricky. A vectorising compiler can quite easily handle flat loops that express simple elementwise operations, but nested loops, as well as loops containing conditionals or dependencies on values computed in a previous loop iteration cannot be vectorised by the C compiler. It is, however, possible, to encode these loops such that vectorising compilers are able to produce fast code. The methods used in the Nepal system are based on [11]. As this code requires some extra work, it is not suitable for conventional processors; and so, we need different types of code generation. By producing vectorisable code, the resulting C code can run reasonably efficient on various vector processors. However, as the exact abilities of the vectorising C compilers differ, it is necessary to tune code generation for each processor family if we want optimal performance.

4.3 Structured Communication Library (SCL)

The last component in the implementation of the Nepal system is the library SCL. It provides support for managing nested vectors and a set of parallel operations. Nested vectors are not monolithic data-structures, but SCL offers flat data vectors and so-called *segment descriptors* that describe a vector's shape,

i.e., its nesting structure. For example, the nested vector $[[1,2],[],[3]]$ is represented by the data vector $[1,2,3]$ and a segment descriptor $[2,0,1]$ that contains the lengths of all subvectors.[4] Both, data vectors and segment descriptors can be processed in parallel. This is important, as the segment descriptors of irregular structures can get arbitrarily big.

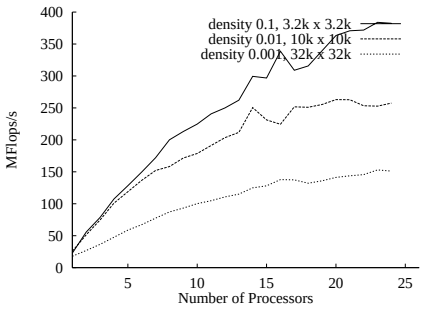
Data vectors and segment descriptors come in two flavours, *balanced* and *unbalanced*. The size of a balanced vector is known to all processors and it is distributed evenly over the available processors. An unbalanced vector consists of a collection of processor-local chunks, one per processor, whose overall size is unknown—each processor only knows the size of its local chunk. Such vectors occur when each processor applies a vector operation whose result size depends not only on the shape, but the actual values of the input vector—for example, filter operations. Even on shared memory architectures, it is most of the time inefficient to immediately rearrange the resulting vector immediately into a balanced vector, hence we need to be able to perform computations on unbalanced vectors. In many cases, this allows us to avoid communication and to fuse more operations into a single local loop. The library provides type conversion operations, from unbalanced into balanced vectors, which explicitly triggers load-balancing.

The second component of the library are the operations manipulating the structure or element order of vectors, like permutations, extraction of elements, concatenations and the like. It might be interesting to note that each of those operations exists in two variants: element and chunk-wise. For example, elementwise permutation is the conventional permutation that might move each element to a new position. Chunk-wise permutation changes the order of whole subvectors. Accordingly, chunk-wise extraction extracts complete subvectors in one step, whereas elementwise extraction can only extract one element at a time. All the chunk operations could be expressed using their elementwise variant. However, this would loose important information, as the block structured communication of the chunk operations can be implemented much more efficiently. The chunk-wise operations proved to be important for the efficient implementation of flattened code.

Finally, we have scan and reduction operations that, given one value per processor, compute the generalised prefix sums and sums, respectively, for operations like $+, *, min, max$, logical *and* and *or*. Note, however, that SCL includes no computations on vectors, like elementwise addition, sum of vectors and so on. They can be built by combining conventional C computations with library functions. This matches with the kernel language, where we have global operations (realized by library functions) combined with sequential operations that are applied in parallel to a collection of values (i.e., the C node programs).

The library is based on a small set of one-sided block-wise and elementwise operations (like *put*, *get*, and *broadcast*). Only their implementation (which accounts for about 5% of the overall SCL code) actually depends on the underlying

[4] In the library, a segment descriptor contains for efficiency reasons more complex information, but here we will just regard them as a vector of lengths.



dense vector-vector multiplication

Machine	T3E (1 PE)	VX/1
MFlop/sec	50.0	823.0

Maximum throughput on big data sets.

smvm on the VX/1

Density	0.1%	0.01%	0.001%
MFlop/sec	105.8	85.6	53.9

Number of non-zero elements approx. 10^6, variation in matrix density.

Fig. 4. Performance of the sparse matrix-vector multiplication (T3E and VX/1)

communication library, therefore, the system is highly portable. Currently, the library works on top of MPI-2 (LAM), Cray's (T3E) SHMEM library, and the Fujitsu VX-1, as well as on sequential machines—the latter, mainly for program development and debugging.

5 Benchmarks

To evaluate the feasibility of the approach, we performed a number of experiments with code generated manually according to the program-transformation rules used in the Nepal system, as the compilation system is not yet fully implemented. In contrast, the SCL library is already largely implemented, so that we used the same runtime system and communication routines that the final system will use. We tested the sparse matrix multiplication code from Section 3 and a hierarchical n-body code based on the Barnes-Hut algorithm.

We ran the sparse matrix code both on the Cray T3E and the Fujitsu VX/1 vector processor for input matrices having different degrees of sparseness. Figure 4 displays the MFlops achieved on both architectures—the numbers include only the multiplication of vector with matrix elements and the summation of these products. As a measure of what both processors are capable, in the upper right corner of the figure, we have given the throughput for a dense vector-vector multiplication (*vvm*). The difference in performance between *vvm* and *smvm*, on a single node, is mainly due to the overhead of handling the sparse and irregular structure in the latter code. On the T3E, *smvm* has a good absolute performance and scales very well for relatively dense matrices (e.g., 10% non-zero elements). The speedup for less dense matrices is worse, but this is to a certain degree a consequence of the growing size of the input vector, which leads to higher communication costs. The performance on the vector processor, while acceptable, should be improved. We had a hard time getting Fujitsu's vectorising C compiler to properly vectorise our code and suspect that—especially for conditionals—it

still generates suboptimal code. It would be interesting to investigate whether the results would improve when we target their Fortran compiler.

Our second code, the Barnes-Hut algorithm [2] computes the forces between a given set a particles; it uses an irregular tree structure to achieve an asymptotically better runtime than the naive $O(n^2)$ force calculation algorithm. We ran the benchmarks for two different types of particle sets, a homogeneously distributed set of particles and a so-called plummer distribution, where the particles center around one point of the area. The Barnes-Hut algorithm requires less computation steps for a homogeneous distribution, as the tree that stores the particles has depth of about $\log n$ for n particles. Roughly speaking, the algorithm has to compute twice as much particle-particle interactions for a set in plummer distribution than for a homogeneously distributed particle set of the same size. The running times for particle sets of 16 000, 24 000, and 32 000 elements are displayed in Figure 5. The curves of the right diagram show the higher absolute runtime of the plummer distribution. The right diagram reveals

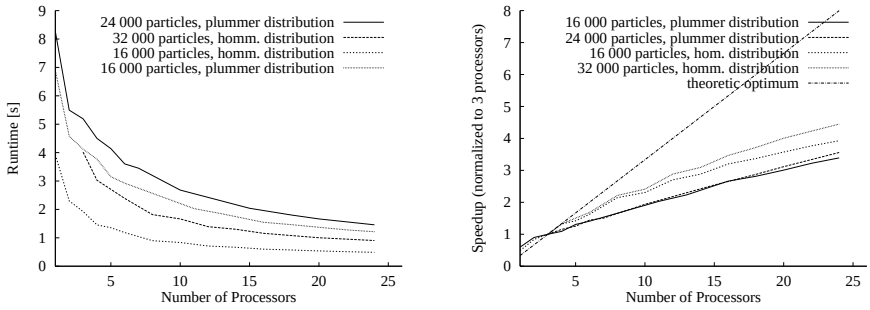


Fig. 5. Runtime and speedup of the Barnes-Hut n-body algorithm on the Cray T3E

another effect: Not only is the absolute runtime of the regular case better, but we also obtain better speed up. On first sight, this might be surprising, as a higher number of computations often leads to programs with better relative speed up. In this case, though, we not only have more computations, but we also have more communication due to the high degree of irregularity. However, the diagram also shows that for 24 processors the speedup for the plummer set is still linear, while it already slows down slightly for the homogeneous sets. Note that the relative speed up is normalised to three processors; this is due to memory constraints, which prevented us from running the big sets on a single node. We like to remark that for a production implementation of n-body codes, there are a number of well-known techniques that improve the parallel behaviour, and in particular, the speedup. One of these techniques is to store more than one particle in each leaf of the tree, which drastically reduces the size of the structure. However, we preferred to avoid using these techniques, to have a stress test for our compiler techniques.

6 Related Work and Conclusions

We already referenced previous work based on the flattening transformation, including our own, throughout this paper. Our contribution here is to consolidate the research in a single portable system. There is a broad scope of research on the parallel implementation of irregular and nested computations in various programming models, too much to enumerate it here. We believe that the use of nested parallelism, in the form outlined in Section 3, in combination with the flattening transformation allows a more portable system than other approaches; however, it remains to be seen whether our system can fulfill this promise. As there are already collective-communication libraries available—in particular, MPI provides such operations—the question arises, why we designed a new one. Our library has two important features: First, it provides memory management for vector structures and directly supports segment descriptors and a number of operations on them; and second, it maps all operations on one-sided communication, which we found convenient to reduce synchronisation points and allow efficient use of shared-memory systems.

Overall, the results so far, encourage use to continue with the chosen approach. Our next steps will include benchmarking our approach on clusters of workstations connected by a high-speed network.

Acknowledgements. The Imperial College Parallel Computing Centre, London, kindly provided us access to their VX/1. We are grateful to Yike Guo, Wolf Pfannenstiel, Jan Prins, and Martin Simons for technical discussions related to the research documented in this paper.

References

1. Peter K. T. Au, Manuel M. T. Chakravarty, John Darlington, Yike Guo, Stefan Jähnichen, Gabriele Keller, Martin Köhler, Wolf Pfannenstiel, and Martin Simons. Enlarging the scope of vector-based computations: Extending Fortran 90 with nested data parallelism. In W. K. Giloi, editor, *Proceedings of the International Conference on Advances in Parallel and Distributed Computing*, pages 66–73. IEEE Computer Society Press, 1997.
2. J. Barnes and P. Hut. A hierarchical $O(n \log n)$ force calculation algorithm. *Nature*, 324, December 1986.
3. Guy E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, 1990.
4. Guy E. Blelloch. NESL: A nested data-parallel language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.
5. Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.
6. Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

7. Guy E. Blelloch and Gary W. Sabot. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing*, 8:119–134, 1990.
8. M. M. T. Chakravarty, F.-W. Schröer, and M. Simons. V—nested parallelism in C. In W. K. Giloi, S. Jähnichen, and B. D. Shriver, editors, *Programming Models for Massively Parallel Computers*, pages 167–174. IEEE Computer Society, 1995.
9. S. Chatterjee. Compiling nested data-parallel programs for shared-memory multiprocessors. *ACM Transactions on Programming Languages and Systems*, 15(3), july 1993.
10. S. Chatterjee, Jan F. Prins, and M. Simons. Expressing irregular computations in modern Fortran dialects. In *Fourth Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Lecture Notes in Computer Science. Springer Verlag, 1998.
11. Siddhartha Chatterjee, Guy E. Blelloch, and Marco Zagha. Scan primitives for vector computers. In *Proceedings of Supercomputing '90*, pages 666–675, 1990.
12. Andrew J. Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In Arvind, editor, *Functional Programming and Computer Architecture*, pages 223–232. ACM SIGPLAN/SIGARCH, ACM, June 1993.
13. C. B. Jay. A semantics for shape. *Science of Computer Programming*, 25:251–283, 1995.
14. Gabriele Keller. *Transformation-based Implementation of Nested Data Parallelism for Distributed Memory Machines.* PhD thesis, Technische Universität Berlin, Fachbereich Informatik, 1999.
15. Gabriele Keller and Manuel M. T. Chakravarty. Flattening trees. In David Pritchard and Jeff Reeve, editors, *Euro-Par'98, Parallel Processing*, number 1470 in Lecture Notes in Computer Science, pages 709–719, Berlin, 1998. Springer-Verlag.
16. Gabriele Keller and Manuel M. T. Chakravarty. On the distributed implementation of aggregate data structures by program transformation. In José Rolim et al., editors, *Parallel and Distributed Processing, Fourth International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS'99)*, number 1586 in Lecture Notes in Computer Science, pages 108–122, Berlin, Germany, 1999. Springer-Verlag.
17. Nepal Project. Nepal compiler & SCL library. `http://www.score.is.tsukuba.ac.jp/~nepal/`, 1999.
18. Y. Onue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system HYLO. In R. Bird and L. Meertens, editors, *Proceedings IFIP TC 2 WG 2.1 Working Conf. on Algorithmic Languages and Calculi, Le Bischenberg, France, 17–22 Feb 1997*, pages 76–106. Chapman & Hall, London, 1997.
19. Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Intl. Conf. on Functional Programming Languages and Computer Architecture*, pages 306–316. ACM Press, New York, 1995.
20. Philip Wadler. *Deforestation: transforming programs to eliminate trees*, volume 73 of *Theoretical Computer Science*. Elsevier Science, 1990.