Optimizing Load Balance and Communication on Parallel Computers with Distributed Shared Memory

Rudolf Berrendorf

Central Institute for Applied Mathematics
Research Centre Jülich
D-52425 Jülich, Germany
r.berrendorf@fz-juelich.de

Abstract. To optimize programs for parallel computers with distributed shared memory two main problems need to be solved: load balance between the processors and minimization of interprocessor communication. This article describes a new technique called data-driven scheduling which can be used on sequentially iterated program regions on parallel computers with a distributed shared memory. During the first execution of the program region, statistical data on execution times of tasks and memory access behaviour are gathered. Based on this data, a special graph is generated to which graph partitioning techniques are applied. The resulting partitioning is stored in a template which is used in subsequent executions of the program region to efficiently schedule the parallel tasks of that region. Data-driven scheduling is integrated into the SVM-Fortran compiler. Performance results are shown for the Intel Paragon XP/S with the DSM-extension ASVM and for the SGI Origin2000.

1 Introduction

Parallel computers with a global address space share an important abstraction appreciated by programmers as well as compiler writers: the global, linear address space seen by all processors. To build such a computer in a scalable and economical way, such systems usually distribute the memory with the processors. Parallel computers with physically distributed memory but a global address space are termed distributed shared memory machines (DSM). Examples are SGI Origin2000, KSR-1, and Intel Paragon XP/S with ASVM [3]). To implement the global address space on top of a distributed memory, techniques for multi-cache systems are used which distinguish between read and write operations. If processors read from a memory location, the data is copied to the local memory of that processor where it is cached. On a write operation of a processor, this processor gets exclusive ownership of this location and all read copies get invalidated. The unit of coherence (and therefore the unit of communication) is a cache line or a page of the virtual memory system. For this reason, care has to be taken to avoid false sharing (independent data objects are mapped to the same page).

There are two main problems to be solved for parallel computers with a distributed shared memory and a large number of processors: load balancing and minimization of interprocessor communication. Data-driven scheduling is a new approach to solve both problems. The compiler modifies the code such that at run-time data on task times and memory access behaviour of the tasks is gathered. With this data a special graph is generated and partitioned for communication minimization and load balance. The partitioning result is stored in a template and it is used in subsequent executions of the program region to efficiently schedule the parallel tasks to the processors.

The paper is organized as follows. After giving an overview of related work in section 2, section 3 gives an introduction to SVM-Fortran. In section 4 the concept of data-driven scheduling is discussed and in section 5 performance results are shown for an application executed on two different machines. Section 6 concludes and gives a short outlook of further work.

2 Related Work

There are a number of techniques known for parallel machines which try to balance the load, or to minimize the interprocessor communication, or both.

Dynamic scheduling methods are well known as an attempt to balance the load on parallel computers, usually for parallel computers with a physically shared memory. The most rigid approach is self scheduling [10] where each idle processor requests only one task to be executed. With Factoring [7]) each idle processor requests at the beginning of the scheduling process larger chunks of tasks to reduce the synchronization overhead. All of these dynamic scheduling techniques take in some sense a greedy approach and therefore they have problems if the tasks at the end of the scheduling process have significantly larger execution times than tasks scheduled earlier. Another disadvantage is the local scheduling aspect with respect to one parallel loop only and the fact that the data locality aspect is not taken into account.

There are several scheduling methods known which have a main objective in generating data locality. Execute-on-Home [6] uses the information of a data distribution to execute tasks on that processor to which the accessed data is assigned (i.e. the processor which owns the data). An efficient implementation of the execute-on-home scheduling based on data distributions is often difficult if the data is accessed in an indirect way. In that case, run-time support is necessary. CHAOS/PARTI [13] (and in a similar manner RAPID [4]) is an approach to handle indirect accesses as it is for example common in sparse matrix problems. In an inspector phase the indices for indirection are examined and a graph is generated which includes through the edges the relationship between the data. Then the graph is partitioned and the data is redistributed according to the partitioning.

Many problems in the field of technical computing are modeled by the technique of finite elements. The original (physical) domain is partitioned into discrete elements connected through nodes. A usual approach of mapping such

problems to parallel computers is the partitioning (e.g. [8]) of the resulting grid such that equally sized subgrids are mapped to the processors. For regular grids, partitioning can be done as a geometric partitioning. Non-uniform computation times involved with each node and data that need to be communicated between the processors have to be taken into account in the partitioning step. The whole problem of mapping such a finite element graph onto a parallel computer can be formulated as a graph partitioning problem where the nodes of the graph are the tasks associated which each node of the finite element grid and the edges represent communication demands. There are several software libraries available which can be used to partition such graphs (e.g. Metis [8], Chaco [5], Party [9], JOSTLE [12]).

Getting realistic node costs (i.e. task costs) is often difficult for non-uniform task execution times. Also, graph partitioners take no page boundaries into account when they map finite element nodes onto the memory of a DSM-computer and thus they ignore the false sharing problem. [11] discusses a technique to reorder the nodes after the partitioning phase with the aim to minimize communication.

3 SVM-Fortran

SVM-Fortran [2] is a shared memory parallel Fortran77 extension targeted mainly towards data parallel applications on DSM-systems. SVM-Fortran supports coarse-grained functional parallelism where a parallel task itself can be data parallel. A compiler and run-time system is implemented on several parallel machines such as Intel Paragon XP/S with ASVM, SGI Origin2000, and SUN and DEC multiprocessor machines.

SVM-Fortran provides standard features of shared memory parallel Fortran languages as well as specific features for DSM-computers. In SVM-Fortran the main concept to generate data locality and to balance the load is the dedicated assignment of parallel work to processors, e.g. the distribution of iterations of a parallel loop to processors.

Data locality is not a problem to be solved on the level of individual loops but it is a global problem. SVM-Fortran uses the concept of processor arrangements and templates as a tool to specify scheduling decisions globally via template distributions. Loop iterations are assigned to processors according to the distribution of the appropriate template element. Therefore, in SVM-Fortran templates are used to distribute the work rather than used to distribute the data as it is done in HPF. Different to HPF, it is not necessary for the SVM-Fortran-compiler to know the distribution of a template at compile time.

4 Data-Driven Scheduling

User-directed scheduling, where the user specifies the distribution of work to processors (e.g. specifying a block distribution for a template), makes sense and

is efficient if the user has reliable information on the access behaviour (interprocessor communication) and execution times (load balance) of the parallel tasks in the application. But the prerequisite for this is often a detailed understanding of the program, all data structures, and their content at run-time. If the effort to gain this information is too high or if the interrelations are too complex (e.g. indirect addressing on several levels, unpredictable task execution times), the help of a tool or an automated scheduling is desirable.

Data-driven scheduling tries to help the programmer in solving the load balancing problem as well as the communication problem. Due to the way data-driven scheduling works, the new technique can be applied to program regions which are iterated several times (sequentially iterated program regions; see Fig. 1 for an example). This type of program region is common in iterative algorithms (e.g. iterative solvers), nested loops where the outer loop has not enough parallelism, or nested loops where data dependencies prohibit the parallel execution of the outer loop. The basic idea behind data-driven scheduling is to gather run-time data on task execution times and memory access behaviour on the first execution of the program region, use this information to find a good schedule which optimizes communication and load balance, and use the computed schedule in subsequent executions of the program region.

For simplicity, we restrict the description to schedule regions with one parallel loop inside, although more complex structures can be handled. Fig. 1 shows a small example program. This program is an artificial program to explain the basic idea. In a real program, the false sharing problem introduced with variable `a` could be easily solved with the introduction of a second array. In this example, the strategy to optimize for data locality and therefore how to distribute the iterations of the parallel loop to the processors might differ dependent on the content of the index field `ix`. On the other side, the execution time for each of the iterations might differ significantly based on the call to `work(i)` and therefore load balancing might be a challenging task. Data-driven scheduling tries to model both optimization aspects with the help of a weighted graph. The compiler generates two different codes (code-1, code-2) for the program region.

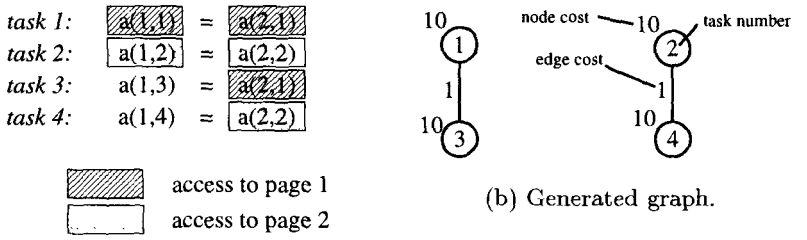
Code-1 is executed on the first execution of the program region[1]. Beside the normal code generation (e.g. handling of PDOs), the code is further instrumented to gather statistics. At run-time, execution times of tasks and accesses to shared variables (marked with the `VARS`-attribute in the directive) are stored in parallel in distributed data structures. Reaching the end of the schedule region, the gathered data is processed. A graph is generated by the run-time system where each node of the graph represents a task and the node weights are the task's execution time. To get a realistic value, measured execution times have to be subtracted by overhead times (measurement times, page miss times etc.) and synchronization times.

To explain the generation of edges in the graph, have a look at Fig. 2(a) which shows the assignments done in the example program. Task 1 and 3 access page 1,

[1] The user can reset a schedule region through a directive in which case code-1 is executed again, for example if the content of an index field has changed.

```
C     --- simple, artificial example to explain the basic idea
C     --- a memory page shall consist of 2 words
      PARAMETER (n=4)
      REAL,SHARED,ALIGNED a(2,n)       ! shared variable
      INTEGER ix(n)                    ! index field
      DATA /ix/1,2,1,2
C     --- declare proc. field, template, and distribute template ---
CSVM$ PROCESSORS:: proc(numproc())
CSVM$ TEMPLATE templ(n)
CSVM$ DISTRIBUTE (BLOCK) ONTO proc:: templ
      ...
C     --- sequentially iterated program region ---
      DO iter=1,niter
C        --- parallel loop enclosed in schedule region ---
CSVM$ SCHEDULE_REGION(ID(1),VARS(a))
CSVM$ PDO (STRATEGY (ON_HOME (templ(i))))
         DO i=1,n
            a(1,i) = a(2,ix(i))      ! possible communication
            CALL work(i)             ! possible load imbalance
         ENDDO
CSVM$ SCHEDULE_REGION_END
      ENDDO
```

Fig. 1. (Artificial) Example of a sequentially iterated program region.



(a) Executed assignments.

(b) Generated graph.

Fig. 2. Memory accesses and resulting graph for example program.

	task				#tasks
	1	2	3	4	
page 1	w/r		r		2
page 2		w/r		r	2
page 3			w		1
page 4				w	1
task time	10	10	10	10	

Fig. 3. Page accesses and task times (w=write access, r=read access).

task 2 and 4 access page 2 either with a read access or with a write access. Page j (j=3,4) is accessed only within task j. Table 4 shows the corresponding access table as it is generated internally as a distributed sparse data structure at runtime. The most interesting column is labeled **#tasks** and gives the number of different tasks accessing a page. If this entry is 1, no conflicts exist for this page and therefore no communication between processors is necessary for this page, independent of the schedule of tasks to processors. If two or more tasks access a page (#**tasks** > 1) of which at least one access is a write access, communication might be necessary due to the multi-cache protocol if these tasks are not assigned to the same processor. This is modeled in the graph with an edge between the two tasks/nodes with edge costs of one page miss. Fig. 2(b) shows the graph corresponding to the access table in table 4 (for this simple example, task times are assumed 10 units and a page miss counts 1 unit).

The handling of compulsory page misses is difficult as this depends on previous data accesses. If the distribution of pages to processors can be determined at run time (e.g. in ASVM[3]) and if this distribution is similar for all iterations, then these misses can be modeled by artificial nodes in the graph which are pre-assigned to processors.

In a next step, this graph is partitioned. We found that multilevel algorithms as found in most partitioning libraries give good results over a large variety of graphs. The partitioning result (i.e. the mapping of task i to processor j) is stored in the template given in the PDO-directive. The data gathering phase is done in parallel, but the graph generation and partitioning phase is done sequentially on one processor. We investigate to use parallel partitioners in the future.

Code-2 is executed at the second and all subsequent executions of the program region and this code is optimized code without any instrumentation. The iterations of the parallel loop are distributed according to the template distribution which is the schedule found in the graph partitioning step.

5 Results

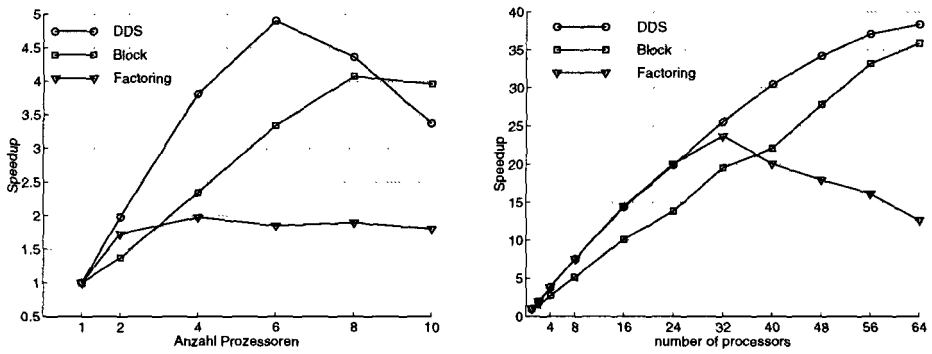
We show performance results for the ASEMBL-program which is the assembly of a finite element matrix done in a sequential loop modeling time steps as it is used in many areas of technical computing. The results were obtained on an Intel Paragon XP/S with the extension ASVM [3] implementing SVM in software (embedded in the MACH3 micro kernel) and on a SGI Origin2000 with a fast DSM-implementation in hardware. A multilevel algorithm with Kernighan-Lin refinement of the Metis library [8] was used for partitioning the graph.

For the Paragon XP/S, a data set with 900 elements was taken, and for the Origin2000 a data set with 8325 elements was chosen. Due to a search loop in every parallel task, task times differ. The sparse matrix to be assembled is allocated in the shared memory region; the variable accesses are protected by page locks where only one processor can access a page but accesses to different pages can be done in parallel. The bandwidth of the sparse matrix is small,

therefore strategies which partition the iteration space in clusters will generate better data locality.

Fig. 4 shows speedup values for the third sequential time step. Because of the relative small size of the shared array and the large page size on Paragon XP/S (8 KB), the performance improvements are limited to a small number of processors.

On both systems, the performance with the Factoring strategy is limited because data locality (and communication) is of no concern with this strategy. Particularly on the Paragon system with high page miss times, this strategy shows no real improvements. Although the shared matrix has a low bandwidth and therefore the Block strategy is in favor, differences in task times cause load imbalances with the Block strategy. Data-driven Scheduling (DDS) performs better than the other strategies on both systems.



(a) Speedup on Paragon XP/S.

(b) Speedup on SGI Origin2000.

Fig. 4. Results for ASEMBL.

6 Summary

We have introduced data-driven scheduling to optimize interprocessor communication and load balance on parallel computers with a distributed shared memory. With the new technique, statistics on task execution times and data sharing are gathered at run-time. With this data, a special graph is generated and graph partitioning techniques are applied. The resulting partitioning is stored in a template and used in subsequent executions of that program region to efficiently schedule the parallel tasks to the processors. We have compared this method with two loop scheduling methods on two DSM-computers where data-driven scheduling performs better than the other two methods.

Currently, we use deterministic values for all model parameters. To refine our model, we plan to investigate into non-deterministic values, e.g. number of page misses, page miss times.

7 Acknowledgments

Reiner Vogelsang (SGI/Cray) and Oscar Plata (University of Malaga) gave me access to SGI Origin2000 machines. Heinz Bast (Intel SSD) supported me on the Intel Paragon. I would like to thank the developers of the graph partitioning libraries I used in my work, namely: George Karypis (Metis), Chris Walshaw (Jostle), Bruce A. Hendrickson (Chaco), and Robert Preis (Party).

References

1. R. Berrendorf, M. Gerndt. Compiling SVM-Fortran for the Intel Paragon XP/S. *Proc. Working Conference on Massively Parallel Programming Models (MPPM'95)*, pages 52–59, Berlin, October 1995. IEEE Society Press.
2. R. Berrendorf, M. Gerndt. SVM Fortran reference manual version 1.4. Technical Report KFA-ZAM-IB-9510, Research Centre Jülich, April 1995.
3. R. Berrendorf, M. Gerndt, M. Mairandres, S. Zeisset. A programming environment for shared virtual memory on the Intel Paragon supercomputer. In *Proc. Intel User Group Meeting*, Albuquerque, NM, June 1995. `http://www.cs.sandia.gov/ISUG/ps/pesvm.ps`.
4. C. Fu, T. Yang. Run-time compilation for parallel sparse matrix computations. In *Proc. ACM Int'l Conf. Supercomputing*, pages 237–244, 1996.
5. B. Hendrickson, R. Leland. The Chaco user's guide, version 2.0. Technical Report SAND95-2344, Sandia National Lab., Albuquerque, NM, July 1995.
6. High Performance Fortran Forum. *High Performance Fortran Language Specification*, 2.0 edition, January 1997.
7. S. F. Hummel, E. Schonberg, L. E. Flynn. Factoring - a method for scheduling parallel loops. *Comm. ACM*, 35(8):90–101, August 1992.
8. G. Karypis, V. Kumar. Analysis of multilevel graph partitioning. Technical Report 95-037, Univ. Minnesota, Department of Computer Science, 1995.
9. R. Preis, R. Dieckmann. *The PARTY Partitioning-Library, User Guide, Version 1.1.* Univ. Paderborn, September 1996.
10. P. Tang, P.-C. Yew. Processor self-scheduling for multiple nested parallel loops. In *Proc. IEEE Int'l Conf. Parallel Processing*, pages 528–535, August 1986.
11. K. A. Tomko, S. G. Abraham. Data and program restructuring of irregular applications for cache-coherent multiprocessors. In *Proc. ACM Int'l Conf. Supercomputing*, pages 214–225, July 1994.
12. C. Walshaw, M. Cross, M.G. Everett, S. Johnson, K. McManus. Partitioning & mapping of unstructured meshed to parallel machine topologies. In *Proc. Irregular 95: Parallel Algorithms for Irregularly Structured Problems*, volume 980 of *LNCS*, pages 121–126. Springer, 1995.
13. J. Wu, R. Das, J. Saltz, H. Berryman, S. Hiranandani. Distributed memory compiler design for sparse problems. *IEEE Trans. Computers*, 44(6), 1995.