Data Dependence Analysis for the Parallelization of Numerical Tree Codes

Gerhard Zumbusch

Friedrich-Schiller-Universität Jena,
Institut für Angewandte Mathematik,
Ernst-Abbe-Platz 2, 07743 Jena, Germany
zumbusch@mathe.uni-jena.de
http://cse.mathe.uni-jena.de

Abstract. Data dependence analysis for automatic parallelization of sequential tree codes is discussed. Hierarchical numerical algorithms often use tree data structures for unbalanced, adaptively and dynamically created trees. Moreover, such codes often do not follow a strict divide and conquer concept, but introduce some geometric neighborhood data dependence in addition to parent-children dependencies. Hence, recognition mechanisms and hierarchical partition strategies of trees are not sufficient for automatic parallelization. Generic tree traversal operators are proposed as a domain specific language. Additional geometric data dependence can be specified by code annotation. A code transformation system with data dependence analysis is implemented, which generates several versions of parallel codes for different programming models.

1 Introduction

Automatic parallelization of general sequential, imperative code is one of the ultimate goals of compiler construction. Numerical algorithms in scientific computing can often be parallelized efficiently with a data parallel approach. Basically there are three related problems to address: First the data partition problem, second the mapping problem where sets of partitions are mapped to a processor, and third the data dependence analysis where to add communication operations in a distributed memory environment and synchronization operations in a shared memory environment. Both partitioning and mapping can be hard problems depending on the data dependence graph. The data dependence analysis itself can be technically too complex for compilers. However, often there exist good solutions for the problems known in the specific area of application. For example additional geometric information along with a geometric partition of data may work very well and may break general NP-hard problems, but such a solution can be impossible to derive solely from a given sequential code. Hence, parallelization of codes written in a domain specific languages may be possible, while parallelization in general is not feasible.

In this paper, we restrict ourselves to hierarchical tree methods, such as fast summation algorithms for N-body simulations and the fast multipole method.

B. Kågström et al. (Eds.): PARA 2006, LNCS 4699, pp. 890–899, 2007.

Given a large number of geometric entities, the numerical algorithms approximate the sum over expensive $N(N-1)/2$ pair wise interactions by combining the action of groups of entities with others distant away. This can be done hierarchically, which leads to an unbalanced k-ary tree. Algorithms work bottom-up for the computation of groups, top-down for the action of groups on other groups or entities and use local neighborhood data on each tree level for pair wise interactions. The overall complexity under reasonable assumptions is reduced to $\mathcal{O}(N\log N)$ or $\mathcal{O}(N)$, depending on the algorithm and the analysis.

There are a number of parallel implementations of such tree algorithms, among them shared [1] and distributed memory implementations [2,3] and references therein. The latter follow a data parallel style with a distributed tree data structures which contains all geometric data. Operations on the tree are subdivided into operations on a common coarse tree part including the tree root and into distributed operations on finer sub-trees exclusively performed and stored on a single processing element. Except for embarrassingly parallel communicationless algorithms, there are algorithms which require top-down or bottom-up data exchange, or some data geometrically close to the tree node. In many cases this can be assembled into a single global communication step and arranged as local tree traversal before or after the communication step. Subtle changes of the numerical algorithm however may cause a more elaborate communication scheme like a data exchange at each tree level or a dynamic process of request and serve processes.

2 Dependence Analysis

A major part for automatic code parallelization is data dependence analysis [4,5]. Efficient distributed memory parallelization however requires global data dependence analysis. There are partial solutions of this problem for loops and array languages like in HPF, parallel libraries [6] and parallel skeletons [7], which have been applied to divide-and-conquer operations [8]. For a detailed overview, see [9].

As an example of a parallel domain language, we consider numerical codes using data organized as one large tree. Algorithms doing so are fast summation techniques like the fast multipole method or some hierarchical grid solver for partial differential equations. Such structures currently cannot be handled automatically in high performance languages and compilers.

The computational atom of the tree algorithms to be discussed is a node data structure. In C++ this is typically a class with some data members and member functions. For tree traversal in the unbalanced tree there are methods to access the child nodes. Furthermore, there may be functions to access nodes in the geometric neighborhood, see Fig. 1. Numerical algorithms on the tree consist of a specification of the (partial) tree traversal along with some operations on the data members of each tree node visited. Of course, there are many different ways to express this [10]. However, for parallelization and for dependence analysis it is favorable to separate tree traversal from operations on the nodes. Even further, it

is often possible to derive the type of tree traversal from the dependence analysis of the operations and to omit the tree traversal code. Hence, tree algorithms can be put into a generic tree library, while the user code specifies the operations on a single tree node. Note that tree creation requires data partitioning and thus requires further domain specific information. Often, some geometric data decomposition can be used, which can also be put into the library.

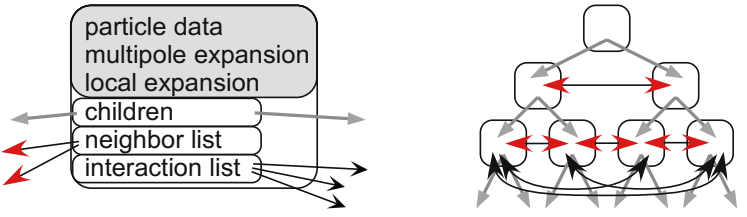

Fig. 1. A sample tree node data structure (left) and the related binary tree (right) for a fast multipole summation

We are left with the data dependence analysis of the operations on a single tree node. A fine scale dependence analysis done by optimizing compilers is not needed here, but solely the relation of data members of the current tree node with other tree nodes. Currently we use a set of m4 scripts to create code, the g++ compiler to process it, and a set of perl script to analyse the code's output. Based on a parallel tree library, this way a dependence analysis of the user code can be performed. The result is a parallel distributed memory message-passing code or a shared memory parallel pThread code or a hybrid message-passing/ pThread code for distributed memory systems with multiprocessor nodes.

A path matrix dependence analysis of the operations on a single tree node is performed, see [5]. The read and write operations on all data members of a node and on all nodes accessed relative to that node are recorded for different stages of the algorithm. A standard bottom-up tree traversal for example will read data of the node and its children and write data of the node, see Fig. 2. Some top-down tree traversal may only write data of the node's children. A detailed comparison of data members read and written reveals flow dependencies which cannot be parallelized this way. Hence, it is possible to verify that some algorithm can be parallelized and to create the actual code for send and receive of the necessary data members at some stage of a parallel tree traversal. Further, the dependence pattern can be used to determine the type of tree traversal and therefore the processor communication pattern. The same information is also used for synchronization operations of a shared memory implementation.

Data dependence analysis on geometric neighbor nodes like in Fig. 2 (right) can be done in the same way as for child/ parent dependencies. However, additional information is needed for the construction of the communication patterns. Given additional user code annotation, for example based on a geometric interpretation of the node relations, a transitive hull is set up, which contains at least

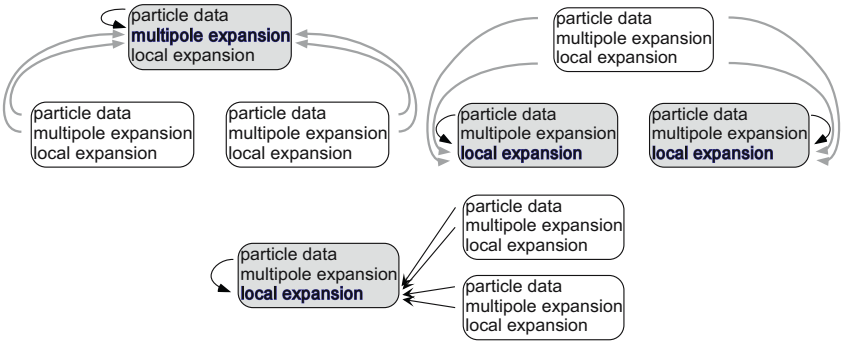

Fig. 2. Sample data dependencies of tree operations: Read/write parent data, read child data, bottom-up traversal (left). Read/write child data, read parent data, top-down traversal (right). Read/write node data, read nodes of interaction list (bottom).

all neighbor nodes involved. We currently use a user defined relation, which may or may not exclude complete sub-trees of interaction nodes. This relation is marked by the code annotation `REQUIRE`, see also the end of the next section.

3 A Fast-Multipole-Method Example Code

As an illustrative example, we provide and discuss some parts of an implementation of a two-dimensional fast-multipole method. The detailed algorithm, formulae and mathematical notation can be found for example in [11]. Here we concentrate on some of the algorithmic structure and features that need to be taken care of in the parallelization. The goal is to evaluate pair interaction forces of a large number of particles. The idea is to approximate long distances computations on coarser parts of a tree of particles, which is done in a bottom-up summation and a top-down evaluation pass.

First of all, we declare a `tree` node derived from a generic k-ary tree defined in the numerical tree library. Multipole and local expansion, particle and field data are added. Further, numerical parts of the algorithm are encapsulated as methods of the `tree` class.

```
class tree : public KAryTree<class tree,4> {
public:
  complex<double> x, field;
  TinyArray1<complex<double>, MAX_EXP_TERMS> mp_exp, local_exp;
  ...
};
```

The tree is created by successive insertion of particles. In order to separate particles from one another, the respective tree node is subdivided. The distributed memory implementation also needs to distribute the tree data structure. A straightforward way to do this is to start with a coarse tree replicated on

all processors. The complete sub-tree of the coarse tree leaf is mapped to exactly one processor. Hence, the tree generation can be performed in parallel. In the case additional load-balancing is necessary, for example space-filling curves can be used [2,3].

```
tree *root = new tree;
```

The first part of the fast multipole summation method computes the far field multipole expansions in a bottom-up order. Children node expansions are shifted to the origin of the parent node expansion and summed up. The implementation consists of some methods of the `tree` classes. We show the declaration only. The comments indicate data dependencies for this presentation. Note that the automatic dependence analysis is based on the actual implementation, not on the comments.

```
  void InitMPExp();              // store mp_exp
  void ComputeMPExp();           // store mp_exp
  void ShiftMPExp(tree *cb);     // store mp_exp, load cb->mp_exp
```

The main code instantiates and uses a tree iterator and contains the actual algorithmic atom to be executed for each tree node. This is a generic tree operator and a first example of the proposed domain language.

```
BottomUpIterator<tree> iu(root);

ForEach(tree *b, iu, ‘
  if (b->isleaf())  b->ComputeMPExp();
  else {
    b->InitMPExp();
    for (int i=0; i<tree::dim; i++)
      if (b->child[i])
        b->ShiftMPExp(b->child[i]);
  }  ’)
```

The code transformation system converts this expression into an ordinary tree traversal in the sequential case. The system is able to determine the type of tree traversal and emits error messages in cases where an unsuitable iterator is specified. However, the construction of a parallel iterator implies that the operations on the children of a node are independent and can be executed in parallel. Hence, for the thread parallel version, sub trees are assigned to different threads, once the coarse tree provides enough sub trees to distribute the load evenly. However, there is some data dependence, namely the child to parent dependence in the array `mp_exp`. This translates into message passing at the level of sub tree to coarse tree on distributed memory machines. The presented code transformation system is able to detect this dependence, even as a inter-procedure code analysis, and to emit the correct message passing instructions.

The second stage of the fast multipole summation computes the interaction lists. For a balanced tree, the interactions are a set of siblings of a node. However,

in the case of unbalanced trees, additional nodes on finer or coarser levels may be needed for the interactions. Nevertheless, the interaction lists can be computed in a top-down tree traversal.

```
TopDownIterator<tree> it(root);
```

For distributed memory machines, we replicate the operations on the already replicated coarse tree, such that no communication or message passing is actually needed in this step.

The final stage of the algorithm, which can also be executed in conjunction with the second stage, computes the local expansions and finally the fields. Here, the far field multipole expansions are evaluated directly or converted into near field local expansions, which need to be evaluated. This can be performed in a top-down tree traversal with a set of methods in the `tree` class,

```
  void VListInter(tree *src);   // store local_exp, load src->mp_exp
  void UListInter(tree *src);   // store field, load src->x
  void WListInter(tree *src);   // store field, load src->mp_exp
  void XListInter(tree *src);   // store local_exp, load src->x
  void ShiftLocalExp(tree *cb); // store cb->local_exp, load local_exp
```

which perform the four types of possible conversions and shift the local expansions for propagation to the children nodes.

```
ForEach(tree *b, it, ‘
  for (int i=0; i<tree::dim; i++)
    if (b->child[i]) {
      b->ShiftLocalExp(b->child[i]);
      b->child[i]->field = 0.0;
      if (b->child[i]->isleaf())
        for (list<tree*>::iterator n = b->child[i]->inter.begin();
             n != b->child[i]->inter.end(); n++) {
          if ((*n)->isleaf()) b->child[i]->UListInter(*n);
          else                b->child[i]->WListInter(*n);
        }
      else
        for (list<tree*>::iterator n = b->child[i]->inter.begin();
             n != b->child[i]->inter.end(); n++) {
          if ((*n)->isleaf()) b->child[i]->XListInter(*n);
          else                b->child[i]->VListInter(*n);
        }
      b->child[i]->EvaluateLocalExp();
    }  ’)
```

Both sequential and thread parallel versions are relatively easy to generate, since there is only a parent to child data dependence in the `local_exp` arrays. However, in the distributed memory version of this code, the dependence on sibling nodes `mp_exp` arrays and particle data `x` turns out to be a severe problem. While it is easy to detect this as a possible dependence, only a global analysis

of the tree and interaction list construction may lead to an efficient and correct result. Some message passing is needed for correctness, but too much may exhaust local memory and degrade parallel efficiency. At this point we clearly see the limitations of automatic parallelization.

The solution we chose here is an additional hint (code annotation) in the application program. The hint provides a criterion to select a set of nodes, which are needed at most. The transformed code initiates a message passing step to exchange the variables determined by the dependence analysis. Data of a node is sent to another processor, if and only if the given criterion may match for any of this processors nodes. Hence, the criterion has to be transitive such that it is fulfilled for a pair of nodes, if it is fulfilled for one of its children and the other node.

```
REQUIRE(list<tree*> neighbor, fetch);
REQUIRE(list<tree*> inter, fetch);
int fetch(tree *b) { return (distance(b) <= 2 * fmin(diam, b->diam)); }
```

Some more details of the code are presented in the following section.

4 Numerical Experiments

For illustration purposes of the concept of generative programming and automatic parallelization of codes written in a domain language we have to implement several systems: First of all, a data dependence analysis tool and a code generation system have to be created. In order to demonstrate its use a domain language and a parallel application library has to be written. Finally a sample application code has to be developed, which is written in the domain language and which is compiled by the code generation system.

The main part of data dependence analysis currently is implemented in a non-robust way leading to a speculative parallelization. The tree atoms of code are compiled and instantiated in different settings. A runtime system keeps track of all references to variables, which results in a dependence analysis capable of interprocedure analysis and recursive calls. However, this requires some programming discipline and the possibility of missing some data dependence. An alternative approach to be pursued in the future would substitute this step by a static code analysis within the optimization phase of a standard C++ compiler. The code generation system further includes multiple passes of the code by the macro preprocessor m4 to generate code, g++ to either compile the code or look for errors and some perl scripts to extract information from the compiler error messages or code instantiations. The results of the compilation process is a correct sequential or parallel code. Each parallel programming model of message passing, thread parallelism or mixed model leads to a different parallel code. The overall execution time of the code generation process is of the same order as standard compilation times of optimizing compilers and are substantially lower than compilation times of some cases of expression templates or self-tuning libraries, see Table 1.

Table 1. Execution times of the source-to-source transformation and compilation times, FMM example, wall clock in sec. Total times are measured and do not exactly match the sum of sub tasks mainly due to pipelining effects of the compilation stages

	sequential	pThreads	MPI	pThreads & MPI
find out of scope variables			9.0	
find local scope variables			1.7	
create instrumented code	–		4.3	
create src code	0.45	0.78	0.75	0.83
compile src, no flags / -O3	2.3/3.2	2.4/3.8	4.8/7.4	5.0/7.4
total, no flags / -O3	11.5/12.6	17.7/20.1	20.2/23.8	21.3/22.9
532 lines of code expand to	590	680	729	777

Now we are ready for the second implementation, namely a numerical tree algorithm to be parallelized and compiled by the code generation system. Again for illustration purposes we chose a well documented example. The fast multipole method in two spatial dimensions can be written for a log$r$ potential conveniently with complex numbers and arithmetic [11] using a Laurent- and a power-series. We chose the fast multipole code of the SPLASH-2 program collection [12] as an initial point. The number of coefficients is fixed both for the far field Laurent series and the near field power series. A quad-tree is constructed with at most one particle per node, each representing a square shaped cell. The particles are distributed uniform over the unit square $[0, 1]^2$ which leads to a slightly unbalanced tree, but good load balance for processor numbers of powers of two.

Wall clock times of a single fast multipole summation, including the computation of far field, near field and the interaction lists are reported on two computer platforms. First, we consider a shared memory computer with four dual-core AMD Opteron processors at 1.8GHz with 64bit Scientific Linux totaling eight processors. In Table 2 we report execution times for two different problem sizes for the sequential, the thread parallel (pThreads), the message-passing (MPI) and the mixed parallel code. Second, the same codes are compiled and run on a beowulf cluster of 32 PCs with a dual-core Intel D820 processor at 2.8GHz running 64bit Rocks Linux connected by gigabit ethernet totaling 64 processors. Three different problems sizes and four programming models are run, see Table 3. The all MPI version places two MPI processes on a computer, while the mixed programming model uses one MPI job, which initiates a second worker thread. On both platforms we use the Mpich MPI implementation with shared memory communication on a computer and p4 device over the network.

On both platforms and for all parallel programming models we observe good parallel speedup and efficiency. Further, four-times the number of particles, leading roughly to four-times the amount of work, also gives good parallel scaling. The most efficient parallel programming model on the shared memory machine is message-passing. The thread implementation is slightly slower. The parallelization is efficient up to eight processors, especially for the larger problem size.

Table 2. Execution times of the FMM example, wall clock in sec. SMP server with 4 dual core processors

no. of proc. cores	1	2	4	8
$0.36 \cdot 10^6$particles, MPI	21.56	10.80	6.47	3.73
2 threads per MPI node		11.60	6.93	3.73
4 threads per MPI node			7.13	4.34
8 threads per MPI node				4.19
$1.44 \cdot 10^6$particles, MPI	84.15	42.11	21.56	11.29
2 threads per MPI node		43.84	24.24	11.68
4 threads per MPI node			25.26	12.73
8 threads per MPI node				13.68

Table 3. Execution times of the FMM example, wall clock in sec. Beowulf cluster with dual core nodes and gigabit ethernet

no. of proc. cores	1	2	4	8	16	32	64
$0.36 \cdot 10^6$particles, MPI	28.42	14.81	7.14	3.45	2.14	2.00	2.73
2 threads per MPI node		15.84	7.64	3.76	2.01	1.30	1.22
$1.44 \cdot 10^6$particles, MPI				15.03	10.17	7.21	5.47
2 threads per MPI node					7.53	3.90	2.70
$5.76 \cdot 10^6$particles, MPI						15.6	8.26
2 threads per MPI node							8.24

For reasons of main memory size, the larger problems can be run on the cluster only for a certain number of computers and above. Up to eight processors, message passing is most efficient. On larger processor numbers the mixed programming model is faster, probably due a limitation of network speed (gigabit ethernet) and a fewer number of communication operations of each computer. Versions up to four processors are executed faster on the AMD processors, but the cluster is faster for eight processors (and more).

Hence all parallel programming models are efficient in general. The optimal choice depends on the platform and number of processors. Execution larger numbers of fast multipole summations, the compilation times (reported from a slower computer) can be neglected.

5 Conclusion

We have discussed some aspects of the automatic parallelization of tree codes. With fast multipole and related algorithms in mind, a programming style with a domain specific tree traversal library and some user code which defines data structures and operations on the tree nodes is briefly introduced. This style allows for a data dependence analysis of the tree algorithms and an efficient

parallelization, in the sense of telescoping programming languages. Code annotation was used, when static dependence analysis was no longer sufficient. The library and preprocessor have been used so far among others for the parallelization of a fast multipole method.

We would like to thank the anonymous referees for their helpful comments.

References

1. Singh, J.P., Holt, C., Gupta, A., Hennessy, J.L.: A parallel adaptive fast multipole method. In: Proc. 1993 ACM/IEEE conf. Supercomputing, pp. 54–65. ACM, New York (1993)
2. Salmon, J.K., Warren, M.S., Winckelmans, G.S.: Fast parallel tree codes for gravitational and fluid dynamical N-body problems. Int. J. Supercomp. Appl. 8(2), 129–142 (1994)
3. Caglar, A., Griebel, M., Schweitzer, M.A., Zumbusch, G.: Dynamic load-balancing of hierarchical tree algorithms on a cluster of multiprocessor PCs and on the Cray T3E. In: Meuer, H.W. (ed.) Proc. 14th Supercomp. Conf., Mannheim, Mateo (1999)
4. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures: A Dependence-Based Approach. Morgan Kaufmann, San Francisco (2002)
5. Hummel, J., Hendren, L.J., Nicolau, A.: A framework for data dependence testing in the presence of pointers. In: Proc. 23rd annual int. conf. parallel processing, pp. 216–224 (1994)
6. Oldham, J.D.: POOMA. A C++ Toolkit for High-Performance Parallel Scientific Computing. CodeSourcery (2002)
7. Kuchen, H.: Optimizing sequences of skeleton calls. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 254–273. Springer, Heidelberg (2004)
8. Herrmann, C., Lengauer, C.: HDC: A higher-order language for divide-and-conquer. Parallel Proc. Let. 10(2/3), 239–250 (2000)
9. Lengauer, C.: Program optimization in the domain of high-performance parallelism. In: Lengauer, C., Batory, D., Consel, C., Odersky, M. (eds.) Domain-Specific Program Generation. LNCS, vol. 3016, pp. 73–91. Springer, Heidelberg (2004)
10. Ananiev, A.: Algorithm alley: A generic iterator for tree traversal. Dr. Dobb's J. 25(11), 149–154 (2000)
11. Beatson, R., Greengard, L.: A short course on fast multipole methods. In: Ainsworth, M., Levesley, J., Light, W., Marletta, M. (eds.) Wavelets, Multilevel Methods and Elliptic PDEs. Numerical Mathematics and Scientific Computation, pp. 1–37. Oxford University Press, Oxford (1997)
12. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., Gupta, A.: The SPLASH-2 programs: Characterization and methodological considerations. In: Proc. 22nd annual int. symp. computer architecture, pp. 24–36. ACM, New York (1995)