Data Parallel Iterators for Hierarchical Grid and Tree Algorithms

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. The data parallel programming language construct of a "for-each" loop is proposed in the context of hierarchically nested arrays and unbalanced k-ary trees used in high performance applications. In order perform an initial evaluation, an implementation of an automatic parallelization system for C++ programs is introduced, which consists of a preprocessor and a matching library for distributed memory, shared memory and mixed model parallelism. For a full compile time dependence analysis and a tight distributed memory parallelization, some additional application knowledge about alignment of arrays or indirect data access can be put into the application's code data declarations. Results for a multigrid and a fast multipole benchmark code illustrate the concept.

1 Introduction

High performance computing should be about a single application of large scale such that both memory size and computing time of a parallel computer limit the precision of the solution computable. A single algorithm operates on a large data set, distributed over the local memories of the parallel processors. Data structures may be uniform or unstructured grids, cells or trees, that is large containers of relatively small, numerical data. It is usually not a good idea to move a substantial amount of data to another local memory or even to redistribute data during computation. Hence a data parallel programming style seems to be natural with operations performed on all elements of the large container. Further, the operations have to operate almost on a local neighborhood only, to be efficiently parallelizable. The "owner computes" paradigm guarantees local memory store operations, such that non-local load operations are the main source of inter process communication.

Parallelization of a sequential code can be done in several steps. First, local and global data dependence analysis can be applied. However, currently they fall short for more complicated data structures and require additional specification [1]. The second step of parallelization is a scheduling and mapping step. Independent operations have to be combined to larger blocks which are mapped to processes or threads. The mapping problem can be far more serious, because again global dependence analysis of the code is required. Once the data structures

W.E. Nagel et al. (Eds.): Euro-Par 2006, LNCS 4128, pp. 625–634, 2006.
© Springer-Verlag Berlin Heidelberg 2006

are instantiated, scheduling and mapping can be written as a large graph problems.However, at compile time the graph is unknown and solutions are available only in very simple cases such as arrays and uniform grids with regular access patterns. This is implemented in numerous data parallel array constructs.

Global dependence analysis of imperative sequential codes is unlikely to solve the scheduling and mapping problem at compile time in general. However, there often is application knowledge, such as geometric properties within a grid or tree, sufficient to enable an efficient parallelization by hand. It is not very economic, however popular to create a full featured parallel programming language for each application area and to incorporate this knowledge. On the other hand, standard library design for common languages is not able to forward this knowledge to an optimizing/parallelizing compiler. Hence there is a current trend to combine library and compiler or some kind of optimizing preprocessor in order to allow for application specific knowledge for parallelization in an abstract programming environment. Such effort include the use of expression templates and extensible source-to-source compilers/optimizers and tools like Rose [2]. A more general concept of application specific code optimization are telescoping languages [3].

The goal of the article is to discuss a preprocessor/library system for parallelization of array and more complex tree data structures common in high performance computing. The sequential programming language is extended by a single data parallel "foreach" construct together with data iterators defined by the library. Data structure dependent parallelization knowledge is confined to the library, application specific parallelization knowledge such as alignment or non-local references can be specified in the application code.

A model implementation uses C++ class libraries and a set of perl scripts, the m4 macro processor and the Gnu g++ compiler to do the local dependence analysis and the source-to-source transformations. Targets currently are distributed memory computers with MPI message passing, shared memory computers with pthreads and hybrid systems with MPI on processes and pthreads to spawn several threads per process. Hierarchies of grids in a multigrid code and unbalanced k-ary trees in a fast multipole code are used to demonstrate the concept. The emphasis of this paper is on the parallel programming style, especially for parallel *tree* algorithms, rather than its model implementation, which can easily be improved. We do acknowledge a large number of alternative solutions for parallel array style programming, which again is not the main subject here. We do *not* advocate the use of scripts and preprocessors for parallel computing, but would like to foster the development of more general and easier ways to incorporate domain specific knowledge into the parallelization of codes. However, even more sophisticated solutions will never be able to perform automatic parallelization of all possible codes.

2 Data Parallel Programming Paradigms

We consider different target architectures. The current preprocessor scans standard C++ code augmented by the "foreach" construct and emits multi-threaded

code for shared memory computers with pthreads, message passing code for distributed memory computers with library calls based on MPI 1, and a combination of message passing between processes and multi-threading within. The message passing calls and parts of the data iterators are encapsulated in a C++ run-time library.

The strategy for distributed memory computing is based on the following considerations:

- Distribute large data structures. Each element is mapped to one process (owner), which is the only process to modify it: "owner computes". The mapping is implemented in the library and is application specific.
- Replicate small data structures and small numbers of operations thereon for each process instead of sending data.
- Use as few send and receive operations as possible. transfers.
- Transfer only data necessary. Perform a dependence analysis at compile-time to determine which data needs to be sent.

Basic message passing operations needed are matching point-to-point send/receive and global reduction operations. The overall performance of the parallel code relies on the pre-computed minimal communication pattern compared to dynamic distributed-shared-memory and related techniques.

Shared memory versions with global address space are easier to implement. The parallel iterator on large data structures with static mapping first cuts the data into several pieces of similar size, and then starts a thread which executes the iterator on each piece and finally waits for the threads to finish. Data is partitioned according to memory layout. Each thread is allowed to modify its own piece of data only, with the exception of global reduction of scalars. Such reductions are done locally for each thread, with a final reduction over all threads.

3 Array Operations

For illustration purposes only we begin with well known for-loops and distributed arrays. We use a block distribution and restrict ourselves to the important part of nearest neighbor communication. This occurs for Finite Difference discretizations on cartesian grids. Each element of an array is associated to a grid point which itself represents a geometrical location. The resulting compact difference stencils represent a finite geometric interaction distance between grid points. Hence it is a good idea to decompose the geometric domain for parallelization, which is done by an array block distribution. Of course, a parallel compiler is not able to figure this out without global code analysis. Hence, the application programmer provides the geometric interpretation implicitly for parallelization through the specification of a block distribution.

3.1 Sequential Semantic

We begin with a code snippet in C++ creating a one dimensional grid, an iterator for all interior grid points and two arrays on the grid. The grid is defined by an

index set, the interval $(0, n+1($. Arrays are allocated according to this index set. Further we introduce an iterator on a subset of the grid, here $(1, n($, in order to treat boundary indices separately.

```
int n = 64;
Grid1 *g = new Grid1(0, n+1);
Grid1IteratorSub it(1, n, g);
DistArray1<double> x(g), y(g);
double e = 0.;
ForEach(int i, it,  x(i) += ( y(i+1) + y(i-1) )*.5;  e += sqr( x(i) ); )
```

The "foreach" loop based on the index set of the iterator expands to the sequential code

```
for(int i=1; i<n; i++) {x(i) += (y(i+1) + y(i-1))*.5;  e += sqr(x(i));}
```

but provides the semantic of independent operations, and consists of arbitrary (reentrant) C++ code including nested function calls. The result is guaranteed to be independent of the sequence. For the reduction of the variable `e` this is only true up to floating point rounding. A two dimensional grid example including different iterators inside and on the boundary reads like this. The grid is represented by a set of index tuples, here $(0, n+1(\times(0, n+1($. Iterator `ita` visits all tuples in $(1, n(\times(1, n($ and iterator `ita` all tuples except for $(1, n(\times(1, n($.

```
Grid2 *g2 = new Grid2(0, n+1, 0, n+1);
DistArray2<double> z(g2), a(g2);
Grid2IteratorSub ita(1, n, 1, n, g2);
ForEach(‘int i, int j’, ita, ‘z(i,j) = ( a(i-1,j-1) + a(i+1,j+1) +
                    a(i-1,j) + a(i+1,j) + a(i,j-1) + a(i,j+1) )/6.;’)
Grid2IteratorOutside itb(1, n, 1, n, g2);
ForEach(‘int i, int j’, itb, ‘z(i,j) = 0.;’)
```

Basically, the nesting of the i and j loops and the execution order is not specified. The "foreach" syntax including comma separator and ‘ ’ quotation marks are due to an m4 preprocessor step and could be changed to semicolon and {} brackets for more C style. Of course there are many different ways to express this including array operations.

3.2 Code Analysis

In the current implementation a sequence of preprocessing steps identifies the variables and types used in the "foreach" loop, checks for data and loop dependence (including inter procedure analysis) and issues warnings if the code does not seem to be parallel, emits communication operations such as send/receive and reduce, transforms loop code and finally creates C++ source code. The code can be compiled with a run-time library which provides the implementations of grids, arrays and the remaining parts of the iterator. We briefly comment on some rationales.

Replicated data, i.e. scalars and small data structures allocated on each process can either be read-only (store is disallowed) in a loop or a reduction variable

(simple load is disallowed). In the case of a reduction, special code is created for thread and/or message passing environments.

Distributed data container and iterators ought to match. Assume that the iterator and the arrays involved share the same index space and distribution. Then it is easy to detect the disallowed cases of non-local store (violates the owner-computes-rule), references to elements more distant than direct process neighbors (violates nearest neighbor communication) and loop carried dependence (non-local load together with local store). Only output-dependence for global objects (e.g. file descriptor cout) is allowed with non deterministic output order. Non-local load operations trigger appropriate send/receive message passing code, which is executed prior to the "foreach" loop.

The model can be generalized to non-neighbor communication, which raises the questions of appropriate data distributions. For indirect addressing see the following chapter on trees.

3.3 Arrays of Different Shapes

The computational model for arrays so far can be extended relaxed to arrays and iterators based on different grids. We are aiming at the multigrid application with a set of nested grids to be discussed later. A notation of grid alignment is used which is slightly different than the HPF guarantees a relationship between the distributions of the arrays. In order to use a fixed communication scheme, a finer grid is created aligned to a coarser one using a mapping function. Further, to be able to compute the communication patterns at compile time, the mapping is also passed to the "foreach" loop as a C++ template type.

To be more precise, assume two one dimensional arrays, a base array with integer index space $[n_0, n_1) \subset \mathbb{Z}$ and another, possibly larger array with index space $[m_0, m_1)$. We define a (truncated) affine monotone mapping $\pi : \mathbb{Z} \to \mathbb{Z}$, which can be written as $\pi(i) = \lfloor (i-k)/m \rfloor$ with constant $m \in \mathbb{N}$, $k \in \mathbb{Z}$. Each index $i \in [n_0, n_1)$ of the base array is mapped uniquely to process $p(i) \in \mathbb{N}_0$. A grid $[m_0, m_1)$ is said to be aligned, iff index $j \in [m_0, m_1)$ is mapped to process $p(\pi(j))$. The mapping and an example code, simplified versions of the following application multigrid code's restriction and prolongation operations, looks like this:

```
class fine { public: int map(int i) {return i / 2;} } f;
Grid1 *gf = new Grid1(0, 2*i+1, g, &f);
DistArray1map<double, fine> z(gf);
ForEach(int i, it, x(i)     = z(2*i)*.5 + ( z(2*i-1) + z(2*i+1) )*.25; )
ForEach(int i, it, z(2*i)   = x(i);
                   z(2*i+1) = ( x(i) + x(i+1) )*.5; )
```

The first "foreach" loop triggers a left process fetch for the array `z` on the finer grid, while the second one triggers a right fetch for array `x`, while still being a local store operation under transformation π.

4 Tree Operations

Now we consider the main target of the paper, namely algorithms on tree data structures with the same "foreach" loop syntax. For alternative ways to write tree iterators see [4]. With a fast multipole summation of particle-particle interactions in mind, a k-ary tree represents a hierarchical decomposition of the computational domain with particles at the leafs of the tree according to their geometrical location. The tree can be written as a directed acyclic graph starting from a root node. A useful data partition in distributed memory starts with a coarse sub-tree from the root node which is replicated on each process. The remaining nodes form a forest of trees, with each tree mapped to one process. The mapping may combine trees geometrically or by some graph partitioning scheme, see [5].

4.1 Communicationless Tree Traversal

The following code snippet shows part of the tree declaration, but hides the library's tree implementation.

```
class tree : public KAryTree<class tree, 2> {
public:     // generic binary tree provides  tree* child(int);
  complex<double> m, l, f, x;
... };
tree *root = new tree;
```

Tree creation proceeds by (parallel) insertion of particles or (parallel) sorting according to some partitioning scheme, where algorithms of different types are involved. Geometric domain decomposition, graph partitioning, space-filling curves and other techniques [5] are available to partition the data in an initial step or after a number of (time-) steps e.g. in a particle simulation. We consider iterators for tree traversal only.

```
TopDownIterator<tree> down(root);
ForEach(tree *b, down, b->f = b->l; )
ForEach(tree *b, down, ‘
  for (int i=0; i<2; i++)
    if (b->child(i))  b->child(i)->l += b->l; ’)
```

The order of execution is no longer arbitrary, but partially ordered, in this case top down from root to the leaves, such that lots of parallelism is exposed. Operations on the replicated coarse tree are executed on all processes and operations on the remaining trees are executed by the respective owners. The first "foreach" loop shows a local assignment completely independent of the execution order. The second one performs a local store at the child nodes, such that a strict parent before child order has to be enforced. Different orders of load and store operations which lead to loop carried dependence trigger warning messages of the preprocessor using path matrix dependence analysis [1]. Except for possible global reduction operations no parallel communication is needed.

4.2 Bottom-Up Communication

A bottom up, leaf to root execution order is shown in the next example.

```
BottomUpIterator<tree> up(root);
ForEach(tree *b, up, ‘
  for (int i=0; i<2; i++)
    if (b->child(i))  b->m += b->child(i)->m; ’)
```

Now, the processes first execute the operations on their own sub trees in a children before parent order. A communication step at the replicated coarse tree's leaf nodes is necessary to update them, which currently uses a global message passing gather operation. Afterwards the operations can be executed on all processes on the coarse tree. The preprocessor determines variables to be transferred (here: `m`). The library does the packing and un-packing. Additional communication would be needed for global reduction operations. Non-local store or different load operations leading to loop dependence would again trigger warning messages. The shared memory implementation performs a synchronization step instead of the communication, with coarse tree operations done by a single thread.

4.3 Communication Within a Geometrical Neighborhood

Besides parent to child and children to parent data flow, fast summation techniques also rely on neighborhood data on all tree levels. However, the nodes actually needed are a small fraction of the full tree and are often determined geometrically. Assume that the operation on node i requires data of 'neighbor' node j, which we denote by relation $i \wedge j$. Each fine tree node i is mapped to process $p(i)$. In a communication step, data of nodes $\bigcup\{j \,|\, i \wedge j,\, p(i) = p_1,\, p(j) = p_2\}$ has be send from p_2 to p_1. In order to do this efficiently, a hierarchical hull relation $i \triangleleft j$ is needed with $i \wedge j \Rightarrow i \triangleleft j$ and $i \triangleleft j \Rightarrow \mathrm{parent}(i) \triangleleft j$ and $i \triangleleft \mathrm{parent}(j)$. Using relation $\triangleleft$ on the coarse tree representation of the data partition is sufficient and each process is able to compute a superset of nodes to be transferred. Such relations are available for many tree codes and one might try to construct them based on more abstract specifications [6].

The following statements within class `tree` declaration define the relation $\triangleleft$ `fetch`, which guards all accesses to nodes pointed to by elements of the interaction list `inter`.

```
Require( list<tree*> inter,  fetch );
double x0, x1;
int fetch(tree *b) { return (x0==b->x1) || (x1==b->x0); }
```

Statement 'require' both declares the variable `inter` and attaches the relation `fetch` to it. The following code shows a tree iterator using indirect addressing.

```
ForEach(tree *b, down, ‘
  for (list<tree*>::const_iterator i = b->inter.begin();
       i != b->inter.end(); i++)
    b->l += log(abs(b->x - (*i)->x)) * (*i)->m;  ’)
```

Each process collects all nodes defined by $\triangleleft$ on the coarse tree's leafs and may be needed by another process and forwards these. The preprocessor determines the variables actually needed (here: `x` and `m`) and looks for loop dependencies. The implementation proceeds with the tree traversal. During the creation of the interaction list for example, no data except for the tree structure is needed.

5 Applications

Basically we want to demonstrate the feasibility of the proposed way of expressing parallelism in numerical array and tree codes. It is essential to see some test cases can be written and translated to lower-level parallel code this way. The communication patterns, the number and volume of messages and the placement of thread synchronization points are identical to hand written code based on the same parallelization strategy. Since such a parallelization is known to be efficient for the given applications, we do not explore in detail the scalability for large numbers of processors or different hardware platforms. A direct comparison to a hand written parallelization is not expected to give new insight at this stage of development.

First we consider a test example for hierarchical arrays. The NAS multigrid benchmark code Fapin [7] implements a geometric multigrid $V_{0,1}$-cycle with one post-smoothing step for a Poisson equation on a set of nested three-dimensional cartesian grids with constant coefficients. The Fortran77 code was ported C++ using the distributed array classes and run for larger data sets (fine grid 513^3) than originally conceived. All message passing and multi-threaded timings are reported for an eight-processor (4 dual-core) AMD64 at 1.8GHz with Scientific Linux 4.1 and Gnu g++ compiler 3.4.3 in 64bit address mode binaries, optimization 'O3'. Compared are timings for Mpich (shmem device) and the native pthread library, see Table 1 left. The mapping of MPI processes and shared memory threads onto the four physical processors and their two processing cores is done dynamically by the operating system.

The parallel speedup on eight processor cores indicate that both different strategies, message-passing (shown vertically) and threads (shown horizontally) work almost equally well with slight advantages for the local address space message-passing. This advantage is probably due to a strong processor to MPI-process binding compared to an arbitrary mapping of processors at each synchronization point of the multi-threaded implementation with effects on access to and caching of local memory banks. This seems to outweigh message passing overhead, which is limited to the transfer of a small fraction of all data. Parallel efficiencies are well above 70% and seem to be limited by the memory bandwidth rather than less efficient coarse grid computations. The measured times for two and four processors involved showed larger variations in different runs due to operating system scheduling, with minimum times shown in the table. Without idle processors the effects vanish. Additional slackness due to more jobs than processors does not improve the numbers significantly, with slight improvements for additional message-passing processors.

Table 1. Parallel speedups of the message-passing (> 1 processes), multi-threaded (> 1 threads), and hybrid (> 1 processes and > 1 threads) program versions on 8 processor cores. Multigrid test case on a nested set of 3D arrays (left) and 2D adaptive fast multipole test case (right).

threads per process	1	2	4	8
no. of processes 1	1	1.50	2.44	5.27
2	1.92	2.51	5.58	5.70
4	2.38	5.77	5.57	5.67
8	5.91	5.63	5.76	5.80

	1	2	4	8
1	1	1.85	3.59	6.71
2	1.94	3.63	6.80	2.87
4	3.91	7.64	6.79	4.28
8	7.79	7.76	7.78	7.73

The second test case covers a hierarchical tree algorithm. Based on parts of the two-dimensional adaptive fast multipole C code FMM of the shared memory Splash-2 benchmarks [8], a C++ implementation using the distributed quad-tree was developed. For reasons of simplicity we consider only at most one particle per leaf cell, using a multipole Laurent series and a local polynomial with 20 complex coefficients each. The tree is partitioned on balanced coarse trees both for message-passing and for multi-threading, although the tree populated with $2 \cdot 10^6$ particles is unbalanced and the tree implementation works for arbitrary partitions. Measured are the times of one field evaluation by the fast multipole method. Parallel load-balancing and tree creation like in [5] could be inserted here additionally, at an initial step and after a couple of multipole evaluations, but involve algorithms of different types (e.g. parallel sorting or embarrassingly parallel) not discussed here. A parallel Barnes-Hut algorithm could be implemented similar to the fast multipole method, but is of higher complexity. The timings were made on the eight-processor system like before. The results are in Table 1 right.

We see again efficient parallelization both for message-passing and for multi-threading with larger advantages for local address space message-passing. Extremely high 97% parallel efficiency are obtained for 8 processes and any number of threads per process. Due to irregular memory access patterns, the shared memory version is slightly slower. Again we obtain large variations in measured times for the partially loaded computer with less than 8 jobs. For the 8 thread case in message passing (2 and 4 processes) we obtain some reproducible timing anomalies. The overall parallel efficiency of the tree code is extremely good as to be expected for large trees of this type.

6 Outlook

We have demonstrated that automatic parallelization does work even for hierarchical algorithms and data structures in high performance computing with a parallelization strategy similar to the ones used for parallelization by hand. However, domain or application specific language extensions were necessary, which

in this case were provided by a combination of a source-to-source preprocessor and a dedicated library.

The parallelism detected so far has been used for coarse grain parallelism. Within a job it is currently not used further. A possible extension would be to exploit the dependence analysis also for code optimization of memory access patterns for hierarchical memory, instruction level parallelism, software-pipelining and for techniques like hyper-threading.

The "foreach" loops and data parallel iterators may also be implemented by a fully fledged C++ source-to-source translator instead of the current scripting solution, which would certainly be an improvement. The concept of telescoping languages would include to have the parallel iterator programming style interoperable with other (parallel) libraries and language extensions. Furthermore, we would like to see easier ways to exploit application specific knowledge for the parallelization in the future.

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

References

1. Hendren, L.J., Hummel, J., Nicolau, A.: Abstractions for recursive pointer data structures: Improving the analysis and transformation of imperative programs. In: Proc. ACM SIGPLAN 1992 conf. Programming language design and implementation, ACM (1992) 249–260
2. Quinlan, D., Schordan, M., Yi, Q., de Supinski, B.R.: Semantic-driven parallelization of loops operating on user-defined containers. In: 16th int. workshop LCPC 2003. Volume 2958 of LNCS., Springer (2004) 524–538
3. Kennedy, K., Broom, B., Chauhan, A., Fowler, R., Garvin, J., Koelbel, C., McCosh, C., Mellor-Crummey, J.: Telescoping languages: A system for automatic generation of domain languages. Proc. of the IEEE **93**(3) (2005) 387–408
4. Ananiev, A.: Algorithm alley: A generic iterator for tree traversal. Dr. Dobb's J. **25**(11) (2000) 149–154
5. Zumbusch, G.: Parallel Multilevel Methods. Adaptive Mesh Refinement and Loadbalancing. Teubner (2003)
6. Birken, K.: Semi-automatic parallelisation of dynamic, graph-based applications. In: Proc. Conf. ParCo'97, Elsevier (1998) 269–276
7. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnam, V., Weeratunga, S.K.: The NAS parallel benchmarks. Inter. J. Supercomp. Appl. **5**(3) (1991) 63–73
8. 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, ACM (1995) 24–36