The Integration of Task and Data Parallel Skeletons

Herbert Kuchen[1] Murray Cole[2]

[1] University of Münster, Department of Information Systems
Leonardo Campus 3, D-48159 Münster, Germany
Email: kuchen@uni-muenster.de
[2] University of Edinburgh, Division of Informatics
King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, UK
Email: mic@dcs.ed.ac.uk

Abstract. We describe a skeletal parallel programming library which integrates task and data parallel constructs within an API for C++. Traditional skeletal requirements for higher orderness and polymorphism are achieved through exploitation of operator overloading and templates, while the underlying parallelism is provided by MPI. We present a case study describing two algorithms for the travelling salesman problem.

1 Introduction

The skeletal model of parallelism [Co89,DPP97,BK98,Da93,Da95,KPS94,Sk94] is well established in the literature as a means of raising the level of abstraction at which programming is undertaken, while remaining sensitive to algorithmic issues which constrain performance. In mainstream approaches all interactions between processes must by explicitly coded (whether through message passing or synchronised access to a shared address space). In contrast, the skeletal approach proposes that recurring patterns of interaction and computation be abstracted and provided for the programmer's use as pre-implemented, parameterised components, for example as language constructs or as a library. This paper describes an instance of the library approach.

A number of design issues arise within such a framework. With respect to language model, it is clear that the skeleton concept has a natural home in the functional style, since skeletons are conceptually polymorphic and higher order. Much preceding work has sought to work within such a context. Equally, pragmatics argue strongly that widespread acceptance will only be achieved when the benefits of skeletons are made accessible in mainstream languages. Subscribing to this view, we choose C++ for the first binding to our library. This enables us to use operator overloading and templates (as suggested in [St00]) to achieve a clean presentation of higher orderness, polymorphism and partial application. With respect to parallelism, a widely applicable system must integrate abstractions which arise from the conceptually divergent worlds of task and data parallelism, since these are often both present in real algorithms. The core of this divergence is the treatment of bulk data. Task parallelism encourages a localised, microscopic view in which parallelism over a data structure is expressed in terms of interactions between processes involving components of the data. For example, in SPMD, message passing parallelised bulk data structures typically have no syntactic presence at all - they exist only in the programmer's mental model as the combination of explicitly named local components. In contrast, data parallelism requires a global, macroscopic view and syntax. The library described here addresses the issue by borrowing the *two tier* model of P3L [DPP97]: a computation consists of nested task parallel constructs, in which the atomic tasks may be either sequential or data parallel. The data parallelism is introduced via skeletons manipulating a distributed data structure in the spirit of Skil [BK96a,BK96b,BK96c,BK98]. Although in principle restrictive, we believe that two tiers are sufficient in practice. There are many algorithms in which data parallel components exist within a task parallel (often pipelined) framework. We are not aware of realistic examples in which the reverse holds. Adding the flexibility of further tiers would complicate the implementation for no obvious benefit.

In summary, we have synthesised aspects of previous work on skeletal task parallelism, skeletal data parallelism and higher order imperative programming into a single programmer-friendly framework. Additionally, the branch-and-bound skeleton at the heart of our main example is new. The remainder of the paper is structured as follows. The data and task parallel skeletons of the library are introduced in section 2 and 3, respectively, with section 4 providing a case study of its application to the solution of the travelling salesman problem and demonstrating the integration of task and data parallelism. Section 5 presents some aspects of the implementation, while section 6 offers conclusions and comments on further work.

2 Data Parallel Skeletons

The skeleton library offers data parallel and task parallel skeletons. Let us first consider the data parallel ones. Data parallelism is based on one or more *distributed data structures*. These are manipulated by operations like `map` and `scan` (parallel prefix), which process a data structure as a whole and which happen to be implemented in parallel internally. These operations can be interleaved with sequential computations working on non-distributed data. Conceptually, this style of programming is almost as easy as sequential programming. Communication problems like deadlocks cannot occur. Currently, two distributed data structures are offered by the library, namely:

```
template <class E> class DistributedArray{...}
template <class E> class DistributedMatrix{...}
```

where `E` is the type of the elements of the distributed data structure. Other distributed data structures, such as distributed lists, may be added in the future. By instantiating the template parameter `E`, arbitrary element types can be generated. This shows one of the major features of distributed data structures and their operations. They are *polymorphic.* A distributed data structure is split into several partitions, each of which is assigned to one processor participating in the data parallel computation. So far only block partitioning is supported. Other partitioning schemes will be added in the future. Typical examples of data parallel skeletons are the following methods in class `DistributedArray<E>`:

```
void mapIndexInPlace(E (*f)(int,E))
E scan(E (*f)(E,E))
```

`A.mapIndexInPlace(g)` applies a binary function `g` to each index position i and the corresponding array element $\texttt{A}_i$ of a distributed array `A` and replaces $\texttt{A}_i$ by $\texttt{g(}i\texttt{,A}_i\texttt{)}$. `A.scan(h)` replaces each element $\texttt{A}_i$ by $\texttt{f(A}_0\texttt{,f(A}_1\texttt{,...f(A}_{i-1}\texttt{,A}_i\texttt{)...))}$, i.e. by the results of successively combining the elements $\texttt{A}_0\texttt{,...A}_i$ with the associative binary function `h`. For example, if `A` consists of the elements with values 1,2,3, and 4 then `A.scan(plus)` replaces them by 1,3,6, and 10 (provided that `E plus(E,E)` adds two elements). The full list of computation skeletons including other variants of `map` as well as different versions of `zip` (which combines corresponding elements of two distributed data structures) and `fold` (also known as `reduce`) can be found in [Ku02a,Ku02b]. It is important to note that skeletons like `map` and `zip` only access the locally available elements on each processor. In order to avoid inefficiency, there is no implicit communication, as occurs for example by accessing elements of remote partitions in HPF [HPF93]. *Communication skeletons* allow exchange of partitions of a distributed data structure between all processors participating in the data parallel computation. Since there are no individual messages but only coordinated exchanges of partitions, deadlocks cannot occur. The most frequently used communication skeleton is

```
void permutePartition(int (*f)(int))
```

`A.permutePartition(f)` sends every partition $\mathtt{A}_{[i]}$ (located at processor i) to processor $\mathtt{f}(i)$. `f` must be bijective, a property which is checked at runtime. Some other communication skeletons correspond to MPI collective operations such as `allToAll`, `broadcastRow`, and `gather`. For instance, `A.broadcastRow(i)` replaces every row of a distributed matrix `A` by row `i`.

Moreover, there are operations which allow access to attributes of the local partition of a distributed data structure, such as `getLocalGlobal`, `getLocalRows`, and `getFirstRow` (see Fig. 1). `A.getLocalGlobal(i,j)` fetches element $\mathtt{A}_{r+\mathtt{i},\mathtt{j}}$ of `A`, where r is the index of the first row of `A` assigned to the local partition. This means `i` is a "local index" (i.e. it refers to the local partition at the calling processor), while `j` is a "global index" referring to the matrix `A` as a whole. Other access operations for all possible combinations of local and global indexing are provided. `getLocalRows` and `getFirstRow` return the number of locally available rows and the (global) index of the first locally available row, respectively. These operations are not themselves skeletons but are frequently used when implementing an argument function of a skeleton.

At first, skeletons like `fold` and `scan` might seem equivalent to the corresponding MPI collective operations `MPI_Reduce` and `MPI_Scan`. However, they are more powerful due to the fact that the argument functions of all skeletons can be *partial applications* rather than just C++ functions. Moreover, they work on user-level data structures directly rather than on low-level buffers. A skeleton essentially defines some parallel algorithmic structure, where the details can be fixed by appropriate argument functions. With partial applications as argument functions, these details can themselves depend on parameters which are computed at runtime.

Consider the code fragment for Gaussian elimination in Fig. 1 taken from [Ku02a]. It assumes that the $\mathtt{n} \times (\mathtt{n}+\mathtt{1})$ coefficient matrix `A` is divided into `p` partitions of `n/p` rows and `n+1` columns, such that processor P_i gets the i-th partition ($i = 0, \ldots, p-1$). Repeatedly the pivot row is broadcast and each processor executes the pivot operation on its partition of `A`. In order to accomplish this, the pivot row is copied to a row of the $\mathtt{p} \times (\mathtt{n}+\mathtt{1})$ matrix `Pivot` (line 16), and this row is then broadcast to every other row of `Pivot` (line 17). Finally, the pivot operation `pivotOp` is applied to every partition of `A` using the skeleton `mapPartitionInPlace` (line 18). For simplicity, it is assumed that the pivot is always non zero. Note that the pivot operation `pivotOp` (line 18) is partially applied to the matrix `Pivot` containing the (copies of the) pivot row as well as to `n/p` (the number of rows of a partition of `A`), to `A.getFirstRow()` (the number of the first row assigned to the local partition of `A`), and to the number `k` of the current iteration. The auxiliary function `curry` is used to transform the C++ function `pivotOp` in such a way that it can be partially applied, i.e. applied to less arguments than it actually needs. Note that `pivotOp` requires five arguments; four of which are provided by the partial application, resulting in a function which needs one more argument. Such a function is exactly what `mapPartitionInPlace` expects as an argument so that it can apply it to the local partition of `LA` of the considered matrix (here `A`). Partial applications are frequently used as arguments of skeletons. Another example is found in line 16, where the function `copyPivot` is partially applied to `A` and `k`. Partial applications are internally represented as data structures, and hence cause a small overhead compared to passing pointers to C++ functions around. The "magic" curry function has been taken from the C++ template library Fact [St00]. Note that we have chosen to use the skeleton `mapPartitionInPlace` here, which manipulates a partition as a whole, rather than `mapIndexInPlace`, which accesses the elements of a distributed matrix one by one.

```
 1 double copyPivot(const DistributedMatrix<double>& A,
 2                  int k, int i, int j, double Pij){
 3   return A.isLocal(k,k) ? A.get(k,j)/A.get(k,k) : 0;}
 4
 5 void pivotOp(const DistributedMatrix<double>& Pivot,
 6              int rows, int firstrow, int k, double** LA){
 7   for (int l=0; l<rows; l++){
 8     double Alk = LA[l][k];
 9     for (int j=k; j<=Problemsize; j++)
10       if (firstrow+l == k) LA[l][j] = Pivot.getLocalGlobal(0,j);
11       else  LA[l][j] -=  Alk * Pivot.getLocalGlobal(0,j);}}
12
13 void gauss(DistributedMatrix<double>& A){
14   DistributedMatrix<double> Pivot(sk_numprocs,n+1,0.0,p,1);
15   for (int k=0; k<Problemsize; k++){
16     Pivot.mapIndexInPlace(curry(copyPivot)(A)(k));
17     Pivot.broadcastRow(k);
18     A.mapPartitionInPlace(curry(pivotOp)(Pivot,n/p,A.getFirstRow(),k));}}
```

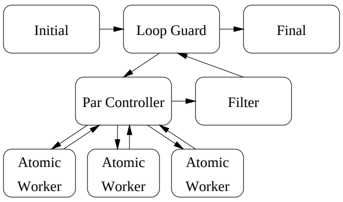
Fig. 1. Gaussian elimination with data parallel skeletons.

The reason is that this allows us to ignore the elements to the left of the pivot column (lines 5-11), while `mapIndexInPlace` would apply its argument function to every element. The argument function would have to behave like the identity function for elements to the left of the pivot column, causing undesired overhead.

3 Task Parallel Skeletons

In some situations there is no distributed data structure which can be manipulated in parallel. Instead, a system of processes communicating via streams of values needs to be built. Our library offers skeletons which allow construction of a process topology by nesting appropriate task parallel skeletons such as pipeline, farm, and parallel composition (see [Ku02b] for the full list of skeletons with detailed specifications).

As an example, let us consider the generation of the Hamming numbers, i.e. numbers which only contain the prime factors 2, 3, and 5. Here, we could for instance construct the topology depicted in Fig. 2. It consists of a pipeline (`p8`) of an initial atomic process (`p1`) generating the initial input, a loop skeleton (`p6`), and a final atomic process (`p7`) printing the result. The loop skeleton passes its inputs to its body (`p5`) and receives the results produced by the body. It has two argument functions `propagate` and `feedback`, which tell for each result, whether it shall be propagated to the next process (here `p7`) and fed back to the body, respectively. The body of the loop consists of a pipeline of a parallel composition (`p3`) and a filter (`p4`). In general, `Loop` non-deterministically merges new inputs with values fed back to the body (although this is incidental to this example since there is only one input value). The parallel composition `p3` consists of three atomic workers (`p2`), which all receive the same inputs and multiply them according to their argument functions by 2, 3, and 5 respectively. The corresponding results are merged by the filter (`p4`) and delivered in ascending order without duplicates back to the loop controller. Note that the filter really gets the expected three values and that there are no problems caused by messages passing each other, since the Par controller only accepts new inputs if all workers have successfully delivered their results. In contrast to an (ordinary) atomic process, a filter does not apply a function to each input value, but it may produce an arbitrary number of output values for each input. To achieve this, the argument function (here `merge`) is called only once. In the implementation of this function, the auxiliary operations `get` and `put` are used as shown in



```
#include "Skeleton.h"
static int current = 0;

int times(int x, int y){return x * y;}
bool lth(int x, int y){return x < y;}
bool propagate(int x){return true;}
bool feedback(int x){return x < 99;}
void merge(Empty dummy){...}

int* init(Empty dummy){if (current++ < 1) return &current;
                       else return NULL;}
void fin(int n){cout << "hamming number: "<< n << endl << flush;}

int main(int argc, char **argv){
   InitSkeletons(argc,argv);

   // step 1: create a process topology (using C++ constructors)
   Initial<int>       p1(init);
   Process*           p2[3];
   p2[0] = new  Atomic<int,int>(curry(times)(2),1);
   p2[1] = new  Atomic<int,int>(curry(times)(3),1);
   p2[2] = new  Atomic<int,int>(curry(times)(5),1);
   Par<int,int>       p3(p2,3);
   Filter<int,int>    p4(merge,1);
   Pipe               p5(p3,p4);
   Loop<int>          p6(p5,propagate,feedback);
   Final<int>         p7(fin);
   Pipe               p8(p1,p6,p7);

   // step 2: start processes
    p8.start();
    TerminateSkeletons();
}
```

Fig. 2. Task parallel example application.

Fig. 3 in order to fetch an input and deliver an output respectively. The farm skeleton (not shown in the above example) works in the usual way. A farmer process accepts a sequence of inputs and assigns each of them to one of several workers. Moreover, the library contains different skeletons for searching, such as the branch and bound skeleton described in the next section. While the b&b skeleton requires a heap for maintaining search problems, skeletons for depth first and breadth first search use a stack and queue respectively.

Each task parallel skeleton has the same property as an atomic process, namely it accepts a sequence of inputs and produces a sequence of outputs. This allows the task parallel skeletons to be arbitrarily nested.

Task and data parallel skeletons can be combined according to the two tier model taken from P3L [DPP97]. This means that on the outer level, we have task parallel skeletons, while data parallel skeletons may be used inside an atomic task parallel process. The case study in the next section shows how this works.

4 The Integration of Task and Data Parallel Skeletons — A Case Study

In order to demonstrate the library we have made two implementations of a well-known parallel solution to the travelling salesman problem (TSP) based on Quinn's presentation

```
void merge(Empty dummy){
  Heap<int> h(lth);
  int* x; int* y; int* z; int min; int current = 0; int val;
  do{x = get();
     if (x != NULL){ min = *x;
       y = get();  if (*y < min) min = *y;
       z = get();  if (*z < min) min = *z;
       h.insert(*x);  h.insert(*y);  h.insert(*z);
       while ((! h.isEmpty()) && (h.top() <= min)){
         val = h.get();
        if (val > current){current = val;  put(val);}}}
  } while (x != NULL);
}
```

Fig. 3. Task parallel example application.

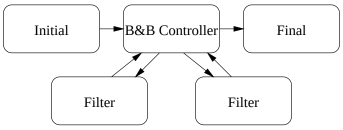
[Qu94] of the parallelisation of Little's algorithm [Bo82,LI63]. This is a straightforward instance of branch & bound. A problem is described by its residual adjacency matrix, a set of chosen edges representing a partially completed tour and a lower bound on the length of any full tour which could be generated by extension of the given partial tour. New problems are produced by a step which

1. selects a "key edge", based on the effect that the exclusion of this edge from prospective tours will have on the lower bound;
2. generates two new problems, in which the chosen edge is respectively included or excluded from the emerging tour, with corresponding amendments to the adjacency matrix and lower bound.

The initial problem consists of the adjacency matrix of the input graph reduced by an operation which generates a lower bound with the observation that any tour must enter and leave each vertex once. There are several potential sources of parallelism:

1. at the highest level, the branching operation (called `generate_cases` in our code) can be applied to many problems concurrently. This is the form of parallelism captured by the `BranchAndBound` skeleton;
2. within `generate_cases`, there is scope for task parallelism between the generation of the "include" and "exclude" sub-problems;
3. in choosing the key edge upon which to branch, there is scope for considerable data parallelism in processing of the adjacency matrix, including both row and column-wise operations and a final matrix-wide reduction;
4. when generating the sub-problems, there is potential data parallelism in the "include" case (amending the whole residual adjacency matrix) and to a lesser extent the "exclude" case (since amendments can be confined to a single row and column).

In our experiments we have developed two programs, one of which exploits only the task parallelism of source 1, the other which extends this with the data parallelism of source 3. Before describing these in more detail, we should highlight the fact that the library's clean integration of task and data parallelism made this incremental approach to parallelisation very straightforward.



```
InitSkeletons(argc,argv);
Initial<problem>         p1(get_problem);
Filter<problem, problem> p2(generate_cases,1);
BranchAndBound<problem>  p3(p2,2,better_than,is_solution);
Final<problem>           p4(& display_results);
Pipe                     p5(p1,p3,p4);
p5.start();
TerminateSkeletons();
```

Fig. 4. Main program pipeline

4.1 Program A: Task Parallelism only

Our `problem` type is a conventional C++ structure combining the information described above. The program is structured as a three stage pipeline (Fig. 4). The first and last stages deal with the file system and are internally sequential, while the second stage calls the branch and bound skeleton. Its test arguments `better_than` and `is_solution` are straightforward and sequential, but more interestingly its main body is a replicated `Filter` process. The replication factor as illustrated is hard wired to 2, but the power of our abstraction makes it a simple matter to vary this with no other changes to the source. We choose a filter because each problem may return either zero (if no key edge can be found) or two new sub-problems. The filtering function `generate_cases` (Fig. 5) performs the "branching" by requesting a problem from the queue, heuristically choosing an edge and generating two new problems, one in which the chosen edge is added to the candidate tour, the other in which it is explicitly excluded.

```
void generate_cases (Empty dummy){
  problem *cand, *with, *without;
  touredge *key_edge;

  do {
    cand = get<problem>();
    if (cand && (cand->solution.length < N) ) {
        key_edge = pick_edge(cand);               // pick edge to branch on
        if (key_edge) {
          with     = add (key_edge, cand);        // case using the edge
          put<problem>(*with);
          without  = exclude (key_edge, cand);    // case omitting the edge
          put<problem>(*without);
        }
    }
  } while (cand != NULL);
}
```

Fig. 5. The filtering function `generate_cases`.

In turn, `generate_cases` calls an auxiliary function `pick_edge`, sketched in Fig. 6, to select the key edge for branching. Its core is a triply nested loop which, for each position in the adjacency matrix performs reductions across the other elements of the corresponding row and column to find values called "`alpha`" and "`beta`" for the given edge, and maintains a "best edge so far" (based on a combination of `alpha` and `beta` values) whose final value is returned. The reductions to find `alpha` and `beta` are associative and commutative. Since it is called from within a `Filter`, `generate_cases` uses `get()` and `put()` to interact with the skeleton. Note that although `get()` blocks until it receives a value, deadlocks are impossible since processes cannot wait cyclically for each other. However, the current implementation of our system can crash, if the processor acting as branch & bound controller runs out of storage (for subproblems).

```
touredge *pick_edge (problem *p) {
  ......
  for (row = 0; row<N; row++) {
    for (col = 0; col<N; col++) {
      .....
      for (i=0; i<N; i++) {
         .... work across this row and column
         .... finding alpha and beta
      }
      ... is this the best edge so far?
    }
  }
  return best edge so far;
}
```

Fig. 6. Sequential `pick_edge`

4.2 Program B: Introducing Data Parallelism

Our second program introduces data parallelism to the `pick_edge` function (Fig. 7). The only change required to the main program is to set the final parameter in the creation of the `Filter` to the number of processes required internally to each of the new data parallel `Filters`. The two outermost loops of the sequential version are replaced by data parallelism (in other words we compute `alpha` and `beta` for all edges concurrently), while the innermost loop becomes a sequence of two one level loops, dealing with the calculation of `alpha` (across rows) and `beta` (down columns) respectively. To achieve this, we introduce two temporary distributed matrices: `Incs` to accumulate the `alpha`, `beta` and related data and `Edges` to contain static global information about the adjacency matrix. These are constructed from the `problem` parameter. We use the `DistributedMatrix` rotation primitives to circulate rows and columns of `Edges` in systolic style and one of the `zipWith` primitives to implement the update at each pulse. The commutativity of the update operation is necessary for correctness. Finally we select the best edge in a single step with a reduction across `Incs`.

4.3 Observations and Performance

Program development was straightforward and allowed the incremental introduction of parallelism, a property which is often held to be pragmatically important to the wider programming community. Program B was developed by a self-contained re-implementation of a single function from Program A, which was itself developed from an initial sequential solution by replacing hand written sequential branch & bound queue coordination code with calls to `Filter` and `BranchAndBound`. We did not implement a hand-coded C++/MPI program, but it is worth noting that this would have had to handle both the inherent non-determinism and termination of the branch and bound queue and the dynamic data distributions of the generated matrices for data parallel processing.

```
touredge *pick_edge (problem *p){
  ....
  DistributedMatrix<edge_tuple> Edges (N, N, curry(mkedge)(p), 2, 2);
  DistributedMatrix<inc_tuple>  Incs  (N, N, curry(mkinc)(p),  2, 2);

  for (i=0; i<N-1; i++){
    Edges.rotateRows(& one);   /* function returning 1 */
    Incs.zipWithIndexInPlace(Edges, & setalpha);
  }
  Edges.rotateRows(& one);      /* Back to where they started */

  for (i=0; i<N-1; i++){
    Edges.rotateCols(& one);
    Incs.zipWithIndexInPlace(Edges, & setbeta);
  }
  Incs.mapInPlace (& setinc);
  best_edge = Incs.fold(& bestinc);
  return the best edge;
}
```

Fig. 7. Data parallel pick_edge

With respect to the details of the library itself, we make a number of subjective observations:

1. the use of a pipeline as the outermost construct is a little unnatural, since we know in advance that the first and last stages are used only once, but is forced upon us by the stream processing model which underpins the task parallel components. Conceptually this need not be a big problem, simply becoming one of the standard programming idioms for the library. Pragmatically, it highlights the need for careful resource management in the implementation, since allocating individual physical processors to these processes would be wasteful, particularly on smaller machines;
2. the branch and bound skeleton fitted the problem naturally;
3. the data parallel primitives allowed compact expression of the matrix manipulations at a level which made the algorithm clear. The provision of true matrices, as opposed to arrays of arrays was helpful, because during the course of its processing the residual matrix must be conceptually treated by row, by column and as a flat aggregate. Thus neither nested casting (i.e. as an array of rows or an array of columns) would have been appropriate.

We have run the programs for a small range of input sizes and number of processors on the Siemens hpcLine machine at the Paderborn Supercomputer Centre. Our experiments produced two conclusions. Firstly, it was clear that the use of data parallelism in program B is counter-productive. To focus on this aspect, we constructed 32 and 64 vertex graphs containing exactly one good tour. Running versions of the programs with a single filter found this tour in the optimal number of steps (ie 32 and 64 respectively). Within this context we compared the run time with a single processor as filter (i.e. program A) against that of a four processor internally data parallel filter (i.e. program B). Program A was faster by factors of around 20 and 40 in the two cases respectively, indicating that for this application, the benefits of data parallelism were hugely outweighed by its communication overheads.

Secondly, considering program A only, running with varying numbers of filters on graphs with many optimal tours, we noted the considerable impact of the race conditions inherent in parallel branch-and-bound on the number of problems examined before a solution was found. This occurred both between “identical” runs (in the sense of having the same data set, program and number of processors) and between runs of the same problem on different numbers of processors. This makes it difficult to compare our absolute performance with that of a plain MPI implementation and we have not attempted to do so. The real point of the case study is to demonstrate that our API makes it easy to experiment with parallel algorithmic structure.

5 Implementation Issues

The implementation of the skeleton library is based on the standard message passing library MPI [GLS99]. Thus, the skeleton library inherits its platform independence from MPI. In order to facilitate portability to other communication libraries we only use the MPI operations for point-to-point communication and refrain from collective operations, since the latter are not necessarily provided by all communication libraries.

5.1 The Implementation of Data Parallel Skeletons

As mentioned earlier, data parallel computations can be interleaved with sequential computations. These sequential computations are replicated (with the same results) on all processors participating in the data parallel computation. This is typically more efficient than executing the computations on one processor and broadcasting the results afterwards. In particular, it implies that map and zip operations require no communication, since they only concern the local partitions and the results of the replicated sequential computations. Fold, scan, and broadcast require a logarithmic number of steps in the number of partitions (for simplicity, counting the computations related to a particular partition as one step). They are based on straightforward variants of the corresponding algorithms found in the literature (e.g. in [Qu94]). Permutation and rotation skeletons require $O(1)$ steps in the number of partitions.

5.2 The Implementation of Task Parallel Skeletons

The skeletons farm, loop, parallel composition, and branch and bound (as well as other search skeletons) use an auxiliary *controller* process to coordinate the corresponding workers. P3L [DPP97] requires that each process produces as many output values as it consumes input values. Since we found this rule too restrictive for practical applications, we have abandoned it. In particular, the filter skeleton can produce an arbitrary number of outputs for each input. Due to this small modification, each controller needs to run a rather sophisticated communication protocol. In particular, it is more difficult to shut down the system cleanly. We use three different tags in order to distinguish messages containing ordinary values, termination test messages, and stop messages. Since a controller cannot rely on a result when it gives some task to a worker (since the worker may be a filter which just discards the task), it sends a termination test message to each worker which has accepted a task (i.e. an ordinary value). Each process forwards a termination test message as soon as it runs out of work. Thus, this termination test message will eventually come back to the controller, which now knows that the corresponding worker is idle and can be supplied with more work. If the initial process runs out of work, it sends a stop message to its successor. Every process forwards a stop message to all connected processes and stops afterwards. Since we shut down the system cleanly after each skeleton-based computation, the main program can spawn an arbitrary sequence of skeleton-based computations. It is also possible to nest skeletons into an arbitrarily nested control structure consisting of C++ loops and conditionals.

5.3 Task and Data Parallelism

According to the two tier model, an atomic task parallel process A may contain data parallelism inside, i.e. it may contain one or more distributed data structures which are manipulated by data parallel skeletons like map and fold. A will typically occupy several processors. On the other hand, A will (as every task parallel process) consume input values from its predecessor and deliver output values to its successor. Since all non-distributed data structures are replicated on all processors assigned to A, we need to make sure that all these processors get the (non-distributed!) input values. In order to achieve this, each input value is broadcast by the specific processor which actually receives the value to all the remaining processors assigned to A. Since all non-distributed data are replicated on all these processors, it does not matter which of them actually delivers the result to the successor. We have chosen to give this obligation to the specific processor which receives the input values, since this simplifies the propagation of termination test and stop messages. All other processors simply ignore output instructions.

6 Related Work

In addition to the general literature on skeletal programming, we draw attention to other work in the areas of integrating task and data parallelism. Of most immediate relevance is $COLT_{HPF}$ [OP99], a run-time support layer for the task parallel coordination of data parallel components which are expressed in High Performance Fortran (HPF), proposed as a means to support both nested skeletal structures (following the conceptual model of P3L [DPP97]), as well as more arbitrary compositions. In contrast to our work, $COLT_{HPF}$ does not propose a specific high level API. More generally, [BH98] discusses a selection of task and data parallel integration projects and identifies three key issues. These are

1. **How to express task and data parallelism.** Bal and Haines observe that existing schemes tend to be biased to one form or the other, depending upon the roots of the language involved. We believe that our approach is relatively neutral in this respect. While our two tier model has tasks encapsulating data parallelism, it is difficult to see what the reverse could usefully mean. More pertinently, the separation of levels allows us to adopt appropriate constructs at each level without compromising either.
2. **How to organise the address space of programs.** Bal and Haines note the tension raised between the multiple address spaces often associated with task parallelism and the shared address space of typical data parallelism. In a sense, the use of skeletons defuses this issue by encapsulating “remote” access within each skeleton’s specification. It is no longer relevant to think of a datum as being within or without a specific address space - it is simply the “next item” to deal with, in a given skeleton’s behaviour. When more flexible manipulation is required within our data parallel layer, our object-oriented approach to distributed matrices is helpful. Its use of communication skeletons recognises the underlying distributed nature of the data but allows it to be handled from shared or distributed perspectives as appropriate.
3. **How to implement task and data parallelism in a single system.** At issue here is the gap between the weighting towards compile-time analysis and scheduling found in data parallel systems and that towards dynamic run-time support more characteristic of task parallelism. To quote [BH98], “If such task parallel constructs are added to a data parallel language, it will make compile-time analysis much more difficult, if not impossible”. Our position here is clear, in that the use of skeletons is intended to constrain matters at the task parallel level in such a way that these too can be handled statically. This opens up interesting possibilities for cross level optimisation which would not be feasible with a more general purpose task parallel layer.

The study of branch-and-bound is a vast field in its own right. Here we simply draw attention to projects which have provided parallel libraries, including [Pa02,SHH95,MA02,LC02]). We differ from these in trying to provide support for branch-and-bound within a wider framework of parallel components.

7 Conclusions and Future Work

We have presented a skeleton library which supports the smooth integration of data parallelism with a rich set of task parallel skeletons. Data parallelism is based on distributed data structures and corresponding skeletons which manipulate them in parallel. Task parallel skeletons set up a system of communicating processes. According to our two tier model an atomic task parallel process may use data parallelism inside. We have demonstrated the integration of task and data parallelism by means of a medium size case study, the solution of the travelling salesman problem based on a variant of Little's branch and bound algorithm. As future work we plan to develop several tools which are tailored to the skeleton library. Among them are a skeleton-based cost analyser and a corresponding optimiser as well as tools for debugging and testing skeleton-based computations. Moreover, we are thinking of extensions to the library, for example dynamic process creation. Within the implementation we intend to consider opportunities for dynamic optimisation of skeleton compositions. The present system simply composes the individual implementations. A more sophisticated scheme could take advantage of the considerable structural information provided by the use of skeletons to customise implementations dynamically.

References

[BH98] H.E.Bal, M. Haines: Approaches for Integrating Task and Data Parallelism, IEEE Concurrency Vol 6(3), pp 74-84, 1998.

[BK96a] G. H. Botorog, H. Kuchen: Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming, in *Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5)*, IEEE Computer Society Press, 1996.

[BK96b] G. H. Botorog, H. Kuchen: Using Algorithmic Skeletons with Dynamic Data Structures, in *Proceedings of IRREGULAR '96*, LNCS, Springer, 1996.

[BK96c] G.H. Botorog, H. Kuchen: Efficient Parallel Programming with Algorithmic Skeletons, Proceedings of Euro-Par 96, LNCS 1123, Springer, 1996.

[BK98] G. H. Botorog, H. Kuchen: Efficient High-Level Parallel Programming, Theoretical Computer Science 196, pp. 71-107, 1998.

[Bo82] T.B. Boffey, Graph theory in operations research, Macmillan, 1982.

[Co89] M.I. Cole: *Algorithmic Skeletons: Structured Management of Parallel Computation*, MIT Press, 1989.

[DPP97] M. Danelutto, F. Pasqualetti, and S. Pelagatti: Skeletons for Data Parallelism in p3l, Euro-Par 97, Springer LNCS 1300, pp. 619-628, 1997.

[Da93] J. Darlington, A. J. Field, P. G. Harrison et al: Parallel Programming Using Skeleton Functions, in *Proceedings of PARLE '93*, LNCS 694, Springer, 1993.

[Da95] J. Darlington, Y. Guo, H. W. To, J. Yang: Functional Skeletons for Parallel Coordination, in *Proceedings of EURO-PAR '95*, LNCS 966, Springer, 1995.

[GLS99] W. Gropp, E. Lusk, A. Skjellum: Using MPI, MIT Press, 1999.

[HPF93] High Performance Fortran Forum: High Performance Fortran Language Specification, Scientific Programming, Vol. 2(1), 1993.

[KPS94] H. Kuchen, R. Plasmeijer, H. Stoltze: Efficient Distributed Memory Implementation of a Data Parallel Functional Language, *PARLE*, LNCS 817, 1994.

[Ku02a] H. Kuchen: The Skeleton Library Web Pages, http://danae.uni-muenster.de/lehre/kuchen/Skeletons/

[Ku02b] H. Kuchen: A Skeleton Library, Report 6/02-I, Angewandte Mathematik und Informatik, University of Münster.

[LC02] B. Le Cun, BOB++, http://www.prism.uvsq.fr/ blec/Research/BOBO/

[MA02] MALLBA Combinatorial Optim. Lib., http://www.lsi.upc.es/ mallba/

[LI63] J. Little, K. Murty, D. Sweeney and C. Karel: An Algorithm for the Traveling Salesman Problem, Operations Research, Vol 11(6), pp. 972-989, 1963.

[OP99] S. Orlando and R. Perego: COLTHPF, a Run-Time Support for the High-Level Coordination of HPF Tasks, Concurrency: Practice and Experience, Vol 11, no 8, pp 407-434, 1999.

[Pa02] University of Paderborn Portable Parallel Branch-and-Bound Library http://www.uni-paderborn.de/fachbereich/AG/monien/SOFTWARE/PPBB/

[Qu94] M. J. Quinn: *Parallel Computing: Theory and Practice*, McGraw Hill, 1994.

[SHH95] Y.Shinano, M.Higaki and R.Hirabayashi: A Generalized Utility for Parallel Branch and Bound Algorithms, 7th IEEE SPDP, pp. 858-865, 1995.

[Sk94] D. Skillicorn: Foundations of Parallel Programming, Cambridge U. Press, 1994.

[St00] J. Striegnitz: Making C++ Ready for Algorithmic Skeletons, Tech. Report IB-2000-08, ZAM, Forschungszentrum Jülich, Germany, http://www.fz-juelich.de/zam/docs/autoren/striegnitz.html