A Framework for Multi-Core Implementations of Divide and Conquer Algorithms and its Application to the Convex Hull Problem *

Stefan Näher   Daniel Schmitt †

Abstract

We present a framework for multi-core implementations of divide and conquer algorithms and show its efficiency and ease of use by applying it to the fundamental geometric problem of computing the convex hull of a point set. We concentrate on the Quickhull algorithm introduced in [2]. In general the framework can easily be used for any D&C-algorithm. It is only required that the algorithm is implemented by a C++ class implementing the job-interface introduced in section 3 of this paper.

1 Introduction

Performance gain in computing is no longer achieved by increasing cpu clock rates but by multiple cpu cores working on shared memory and a common cache. In order to benefit from this development software has to exploit parallelism by multi-threaded programming. In this paper we present a framework for the parallelization of divide and conquer algorithms and show its efficiency and ease of use by applying it to a fundamental geometric problem: computing the convex hull of a point set in two dimensions.

In general our framework supports parallelization of divide and conquer algorithms working on on linear containers of objects (e.g. an array of points). We use the STL iterator interface ([1]), i.e., the input is defined by two iterators $left$ and $right$ pointing to the leftmost and rightmost element of the container. The framework is generic. It can be applied to any D&C-algorithm that is implemented by a C++ class template that implements a certain job interface defined in section 3.

The paper is structured as follows. In Section 2 we discuss some aspects of the parallelization of D&C-algorithms, Section 3 defines the job-interface which has to be used for the algorithms, such that the solvers presented in Section 5 can be applied. Section 6 presents some experimental results, in particular the speedup achieved for different numbers of cpu cores and different problem instances. Finally, Section 7 gives some conclusions and reports on current and ongoing work.

*This work was supported by DFG-Grant Na 303/2-1

†Department of Computer Science, University of Trier, Germany. {naeher, schmittd}@uni-trier.de

2 Divide and Conquer Algorithms

Divide and conquer algorithms solve problems by dividing them into subproblems, solving each subproblem recursively and merging the corresponding results to a complete solution. All subproblems have exactly the same structure as the original problem and can be solved independently from each other, and so can easily be distributed over a number of parallel processes or threads. This is probably the most straightforward parallelization strategy. However, in general it can not be guaranteed that always enough subproblems exist, which leads to non-optimal speedups. This is in particular true for the first divide step and the final merging step but is also a problem in cases where the recursion tree is unbalanced such that the number of open sub-problems is smaller than the number of available threads.

Therefore, it is important that the divide and merge steps are solved in parallel when free threads are available, i.e. whenever the current number of sub-problems is smaller than number of available threads. Our framework basically implements a management system that assigns jobs to threads in such a way that all cpu cores are busy.

3 Jobs

In the proposed framework a *job* represents a (sub-)problem to be solved by a D&C-algorithm. The first (or root) job represents the entire problem instance. Jobs for smaller sub-problems are created in the divide steps. As soon as the size of a job is smaller than a given constant it is called a *leaf* job which is solved directly without further recursion. As soon as all children of a job have been solved the merge step of the D&C-algorithm is applied and computes the result of the entire problem by combining the results of its children.

In this way jobs represent sub-problems as well as the corresponding solutions. Note that the result of a job is either contained in the corresponding interval of the input container or has to be represented in a separate data structure, e.g. a separate list of objects. Quicksort is an example for the first case and Quickhull (as presented in Section 4.3) for the second case.

The algorithm is implemented by member functions of the job class which must have the following interface.

```
class job
{ job(iterator left, iterator right);
  bool  is_leaf();
  void  handle_leaf();
  list<job> divide();
  void  merge(list<job>& L);
};
```

In the constructor a job is normaly created by storing two iterators (e.g. pointers into an array) that define the first and last element of the problem. If the `is_leaf` predicate returns true recursion stops and the problem is solved directly by calling the handle_leaf operation. The `divide` operation breaks a job into smaller jobs and returns them in a list, and the `merge` operation combines the solutions of sub-jobs (given as a list of jobs) to a complete solution. There are no further requirements to a job class.

4 Algorithms

In this section we present job definitions for some well known divide and conquer algorithms. We use Quicksort as an introductive example and then discuss two convex hull algorithms, gift wrapping and Quickhull.

4.1 Quicksort

Quicksort takes as input an array given by the random access iterators `left` and `right`. Functions `merge` and `handle_leaf` are trivial. The `divide` operation calls a function `partition(l,r)` that performs the partition step with a randomly selected pivot element. It returns the position of the pivot element as an iterator m. Finally, it creates two jobs for the two sub-problems.

```
template<class iterator> class qs_job {
 iterator left, right;

public:

qs_job(iterator l, iterator r): left(l),right(r){}
int  size()    { return right - left + 1; }
bool is_leaf() { return size() <= 1; }
void  handle_leaf() {}
void  merge(list<qh_job>& children){}

list<qh_job> divide()
{ iterator m = partition(left,right);
  list<qh_job> L;
  L.push_back(qs_job(left,m));
  L.push_back(qs_job(m + 1,right));
  return L;
}
};
```

4.2 Gift Wrapping

The well-known Gift Wrapping algorithm constructs the convex hull by folding a halfplane around the set of input points such that all points always lie on the same side of the halfplane. In the recursive version of the algorithms two disjoint convex hulls are combined by computing tangents to both hulls. The divide and conquer algorithm is designed as follows:

Partition the input points at some pivot position according to the lexicographical ordering of the cartesian coordinates in two sets L and R, such that the convex hulls of L and R are disjoint. Then compute the convex hull L and R and the extreme points min and max of both hulls recursively. Finally compute the upper and lower tangents starting with line segment $(max(L), min(R))$.

We assume that the input is unsorted and use the Quicksort partitioning step for creating the two subproblems. This gives an expected running time of $O(n \log n)$. Note that we sort the input and compute the convex hull at the same time by exploiting the fact that Quicksort has a trivial merge and Gift Wrapping a trivial divide operation.

The corresponding `gw_job` class is derived from `qs_job`. It inherits the input iterators and the operations `size` and `divide`. The convex hull is stored in a doubly-linked list `result`. The class contains in addition iterators `min` and `max` pointing to the extreme points of the hull. Function `handle_leaf()` treats the trivial case of input size one.

The `merge` operation is illustrated in Figure 1. The auxiliary function `compute_tangents()` does the main work by computing the two tangents as described above.

```
template<class iterator> class gw_job : qs_job
{
 list<point> result;
 list_iterator min, max;

public:

qs_job(iterator l, iterator r): qs_job(l,r){}
void  handle_leaf()
{ if (size() == 1) {
  result.push_back(*left);
  max = min = result.begin();}
}
void  merge(list<qh_job>& children)
{ qh_job jleft = children.front();
  qh_job jright = children.back();
  result = compute_tangents(jleft.result,
      jleft.max,jright.min,jright.result);
  min = jleft.min;
  max = jright.max;
}
```

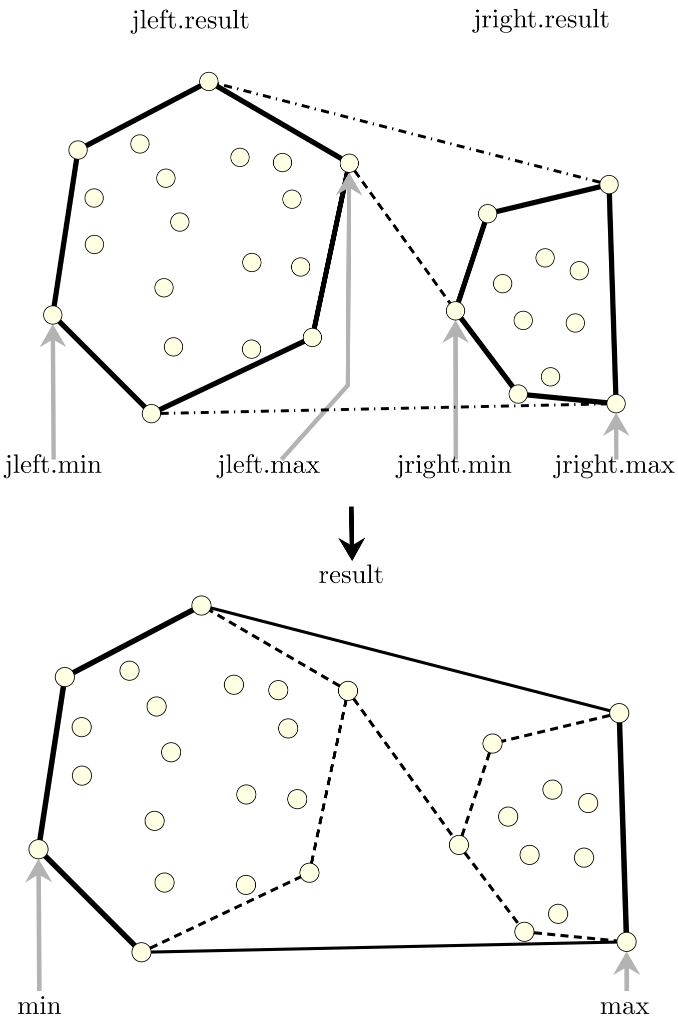

Figure 1: The `merge` operation of Gift Wrapping.

```
};
```

4.3 Quickhull

We show how to define a job class `qh_job` implementing the well-known Quickhull algorithm ([2]) for computing the convex hull of a point set. For simplicity we consider a version of the algorithm that only computes the upper hull of the given point set and we assume that the input is give by a pair of iterators `left` and `right` into an array of points such that `left` contains the minimal and `right` the maximal point in the lexicographical xy-ordering. The result of a `qh_job` instance is the sequence of points of the upper hull lying between left and right. In this scenario any job of size two (only the leftmost and rightmost point) represents a leaf problem and has the empty list as result. Consequently, the handle_leaf operation is trivial (keeping an empty result list).

The `divide` operation is using two auxiliary functions: `farthest_point(l,r)` computes a point between

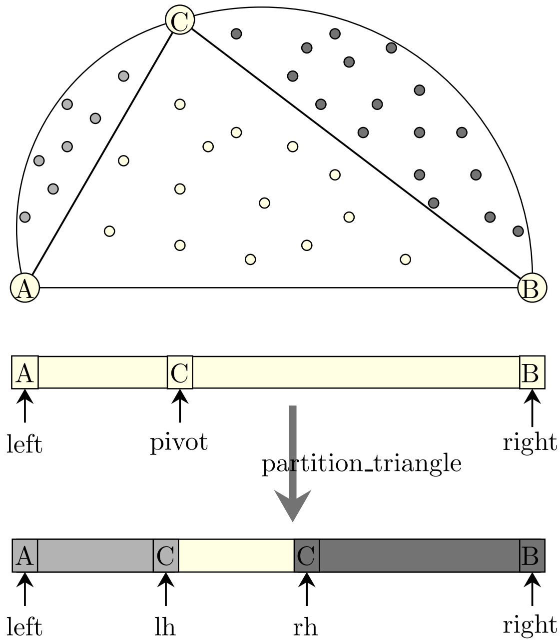

Figure 2: The partition step of Quickhull.

l and r with maximal distance to the line segment (l, r) and `partition_triangle` implements the partition step of quickhull as shown in Figure 2 and returns the generated sub-problems as a list of jobs. We tried different variants of this partition function. In particular, one using only one thread and one using all available threads. The latter version is similar to the parallel partition strategy proposed in [8] for a multi-core implementation of Quicksort. In the experiments in Section 6) we will see that this can have a dramatic effect on the speedup achieved.

Finally, the `merge` operation takes a list of (two) jobs as input, concatenates their result lists, and inserts the right-most point of the first problem in between. The complete implementation is given by the following piece of C++ code.

```
template<class iterator> class qh_job {
 iterator left;
 iterator right;
 list<point> result;

public:

qh_job(iterator l, iterator r): left(l),right(r) {}
int  size()    { return right - left + 1; }
bool is_leaf() { return size() == 2; }
void  handle_leaf() {}

list<qh_job> divide()
{ iterator pivot = farthest_point(left,right);
  iterator lh,rh;
  partition_triangle(pivot,left,right,lh,rh);
```

```
  list<qh_job> L;
  L.push_back(qh_job(left,lh));
  L.push_back(qh_job(rh,right));
  return L;
}

void  merge(list<qh_job>& children)
{ qh_job j1 = children.front();
  qh_job j2 = children.back();
  result.conc(j1.result);
  result.push_back(*j1.right);
  result.conc(j2.result);
}
};
```

5 Solvers

Our framework provides different *solvers* which can be used to compute the result of a job. As a very basic and simple example we give the code for a generic serial recursive solver. It can simply be implemented by a C++ function template.

```
template <class job>
void solve_recursive(job& j)
{ if (j.is_leaf()) j.handle_leaf();
  else { list<job> Jobs = j.divide();
         job x;
         forall(x,Jobs) solve_recursive(x);
         j.merge(Jobs);
        }
};
```

Note that `solve_recursive` is a generic dc-solver. It accepts any job type *job* that implements the `dc_job` interface. We can now use it easily to implement a serial quickhull function taking an array of points as input.

```
list<point> QH_SERIAL(array<point>& A)
{ int n = A.size();
  qh_job<point*> j(A[0],A[n-1]);
  solve_recursive(j);
  list<point> hull = j.result;
  hull.push_front(A[0]);
  hull.push_back(A[n-1]);
  return hull;
};
```

It is an easy exercise to write a non-recursive version of this serial solver: simply push all jobs created by divide operations on a stack and use an inner loop processing all jobs on the stack.

5.1 Parallel Solvers

Parallel solvers are much more complex. They maintain open jobs, build the recursion tree while the algorithm proceeds and check for the mergeability of subjobs. They also have to administrate all threads working in parallel. In our framework all threads use a common job queue which has to be synchronized using a mutex variable.

There are different solver versions according to different requirements. The simplest solver handles problems with a trivial merge step in which case it is not necessary to store the recursion tree explicitely. In our framework solvers distinguishes two types of threads. Primary threads work in parallel in different parts of the recursion tree, and secondary threads parallelize basic operations like partitioning and merging. A solver always tries to employ as much primary threads as possible.

There are more parameters that can be changed by corresponding methods of the class. For instance, a limit d for the minimal problem size for any thread. If a the size of job gets smaller than d it will not be divided into new jobs but solved by the same thread using a serial algorithm. Using this limit the overhead of starting a huge number of threads on very small problem instances can be avoided. We implement parallel solvers by C++ class templates. In the example one sees the interface of a solver class. The constructor takes as argument the number of threads to be used for solving the problem. The computation starts with calling the `run` function with a list of root jobs.

```
template <class Job>
class dc_parallel_solver {
public:
  dc_parallel_solver(int thread_num);
  void set_limit(int d);
  void run(list<Job*> j)
};
```

We now can use the parallel solver template to implement a parallel version of the quickhull function.

```
list<point> QH_PARALLEL(array<point>& A, int thr_n)
{ int n = A.size();
  dc_parallel_solver<job<point*> > solver(thr_n);
  job<point*> j(A[0],A[n-1]);
  solver.run(j);
  list<point> hull = j.result;
  hull.push_front(A[0]);
  hull.push_back(A[n-1]);
  return hull;
};
```

6 Experiments

All experiments were executed on a Linux PC with an Intel quad-core processor running at a speed of 2.6 GHz. As implementation platform we used a thread-safe version of LEDA ([6]). In particular, we used the exact geometric primitives of the rational geometry kernel and some of the basic container types such as arrays and lists. All programs were compiled with gcc 4.1.

In the Quicksort experiments, we sort arrays of integers of various size. Our Quicksort implementation uses a parallel partition function.

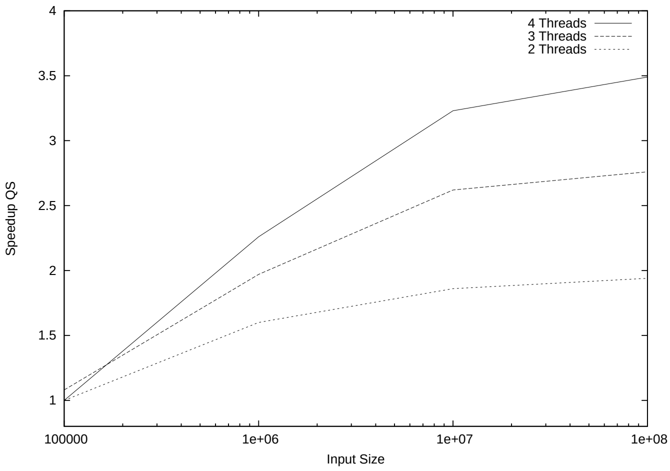

Figure 3: Quickhull: Speedup of Quicksort.

In figure 3 we see the speedup growing near optimal values when the size of the input gets large. To achieve a benefit from parallization we need sufficient problem sizes.

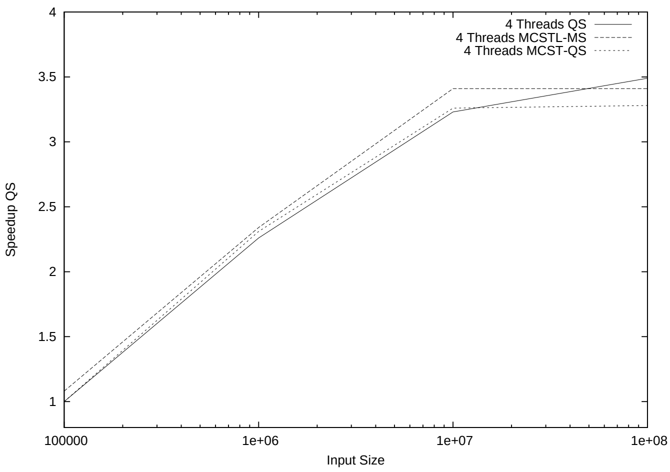

Figure 4: Quickhull: Comparison Quicksort with MCSTL.

Figure 4 compares our Quicksort implementation with two implementations form the MCSTL library ([7]). The MCSTL implementation with the better speedup is based on Mergesort the other on Quicksort. All implementations have a very good speedup progress. However the best speedup not always corresponds to the best absolute running time. Table 1 shows the running times of an input of 10^8 integers and different thread numbers. Our implementation is marginal faster.

	1	2	3	4
QS	10.31	5.32	3.74	2.96
MCSTL-MS	10.47	5.45	3.94	3.07
MCSTL-QS	10.47	5.92	4.02	3.19

Table 1: The Table shows running times in seconds of three Sorting Algorithms. On the x-axis is the number of threads. The input was 10^8 integers.

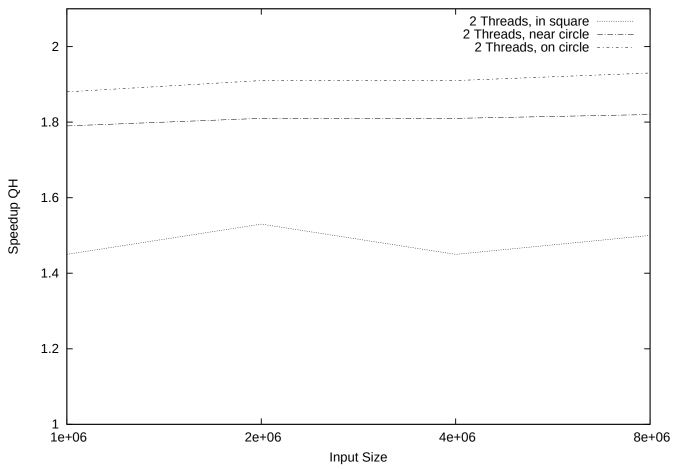

Figure 5: Quickhull: Speedup with 2 cores.

For the convex hull experiments we used three different problem generators: random points lying in a square, random points near a circle, and points lying exactly on a circle. Figures 5 and 6 show that we our framework achieved a good speedup behavior for points on or near a circle, which is the difficult case for Quickhull because only a few or none of the points can be eliminated in the partitioning step. Note that the 1.0 baseline indicates the performance of a serial version of the algorithm (using only one thread). It turned out that $n/100$ was good choice for the limit mentioned in section 5.1.

For random points in a square Quickhull eliminates almost all of the input points in the root job of the algorithms (with high probability), i.e. almost the entire work is done here. In this case the achieved speedup is not optimal. However, Figure 7 shows that without parallelization of the partitioning step we have no speedup at all. We have some ideas to improve the parallel partitioning and hope to improve the results for this kind of problem instances.

We also want to mention here that we ran experiments with different D&C algorithms for convex hulls. In particular, a recursive version of the gift wrapping method where the merge step does most of the work by constructing two tangents. Figure 8 shows the speedup behavior of this algorithm for the same set of input instances.

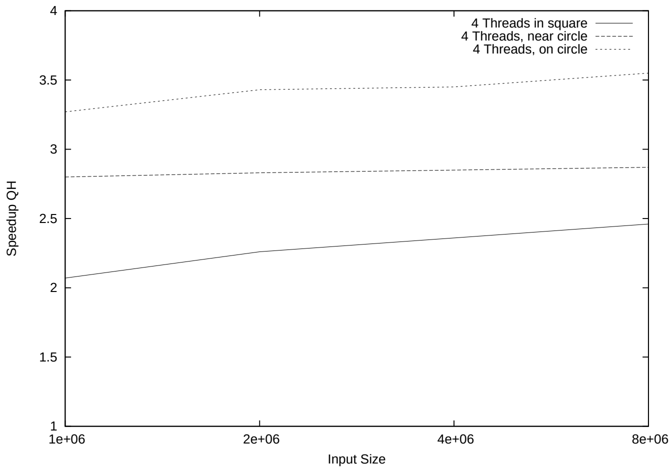

Figure 6: Quickhull: Speedup with 4 cores.

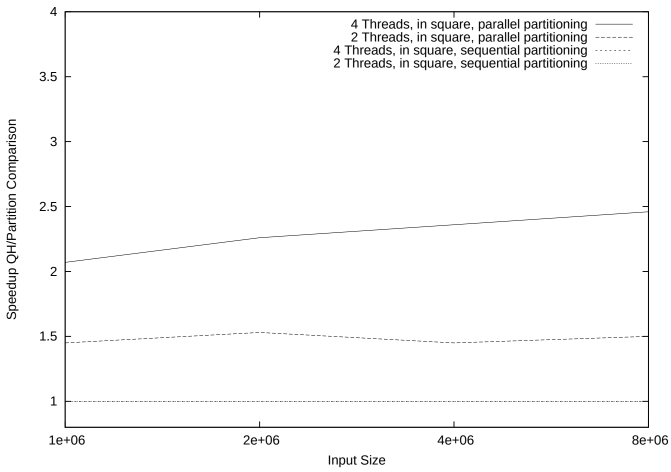

Figure 7: Quickhull: The effect of parallel partitioning.

Furthermore we used two additional serial solvers for the gift wrapping job. One implements the random incremental construction algorithm (RIC) of M. Kallay [4], the other Grahams Scan (GS) [5]. Both show a very good spedup for points on a circle and worse speedup for points in a square compared to the serial solver of Gift Wrapping, see 9.

However the absolute running times show in all cases a better performance of the GS solution. The serial random incremental solver is even better for small convex hulls (points in square) and really worse for large results, which is a characteristic of this algorithm. Some experimental results are given in the tables 2 and 3.

A serial algorithm can improve the performance of a D&C implementation by solving subproblems more efficient. The main point of D&C stays the creation of a sufficient number of subproblems. Note that the Graham Scan Algorithm of our example is inherently sequential and therefore cannot be parallelized easily.

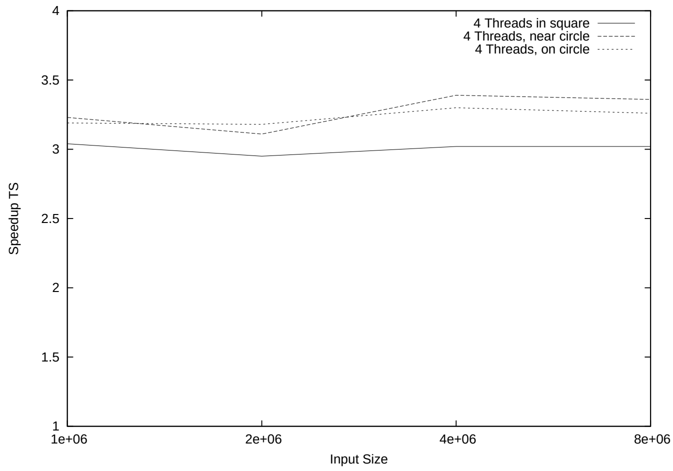

Figure 8: Gift Wrapping: Speedup with 4 cores.

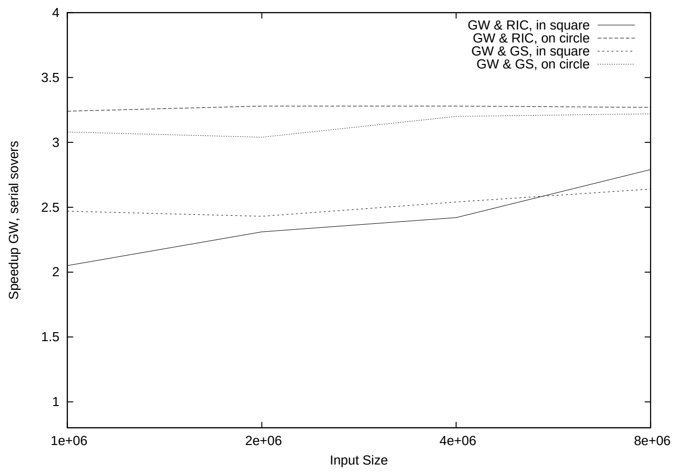

Figure 9: Gift Wrapping: Speedup with 4 cores and different serial solvers.

	$1*10^6$	$2*10^6$	$4*10^6$	$8*10^6$
GW	0.69	1.54	3.3	7.26
GW & RIC	0.4	0.78	1.72	3.26
GW & GS	0.49	1.14	2.67	6

Table 2: Gift Wrapping/points in square: The Table shows running times in seconds of the Gift Wrapping Algorithm with three different serial solvers. On the x-axis is the size of input.

7 Conclusions

We have presented a framework for the implementation and parallelization of divide and conquer algorithms. The framework is generic (by using C++ templates) and can be used very easily. The experiments show that a considerable speedup can be achieved by using two or four threads on a quad core machine. We have some ideas to improve the parallel partitioning of the

	$1*10^6$	$2*10^6$	$4*10^6$	$8*10^6$
GW	1.26	2.78	5.85	12.66
GW & RIC	1	2.31	4.98	10.79
GW & GS	2.15	6.19	17.49	46.42

Table 3: Gift Wrapping/points on circle:The Table shows running times in seconds of the Gift Wrapping Algorithm with three different serial solvers. On the x-axis is the size of input.

quickhull algorithm and hope to be able to improve the efficiency in cases where most of the work is done in the root job. In this paper we presented a sample of our experimental results with different D&C Algorithms. We also work on the parallelization of incremental algorithms for geometric problems and higher dimensional problems. One of the major problems is the need of more complicated thread-safe dynamic data structures such as graphs or polyhedra.

References

[1] M. H. Austern *Generic programming and the STL*, Addison-Wesley, 2001.

[2] A. Bykat *Convex hull of a finite set of points in two dimensions*. IPL, 7:296-298, 1978.

[3] W. F. Eddy *A new convex hull algorithm for planar sets*. ACM Trans. Math. Softw. 3, 1977.

[4] M. Kallay *The Complexity of Incremental Convex Hull Algorithms*. Info. Proc. Letters 19, 197, 1984

[5] R.L. Graham *An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set*. Information Processing Letters 1, 132-133 (1972)

[6] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[7] Felix Putze, Peter Sanders, Johannes Singler *MCSTL: The Multi-Core Standard Template Library*. Technical Report, MCSTL Version 0.70-beta

[8] Philippas Tsigas and Yi Zhang *A Simple, Fast Parallel Implementation of Quicksort and its Performance Evaluation on SUN Enterprise 10000.*