

# Practical Parallel Divide-and-Conquer Algorithms

Jonathan Hardwick

Committee:

Guy Blelloch (chair)

Adam Beguelin

Bruce Maggs

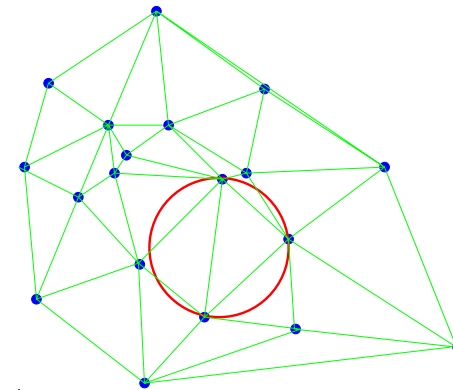
Dennis Gannon (Indiana University)

# A motivating problem

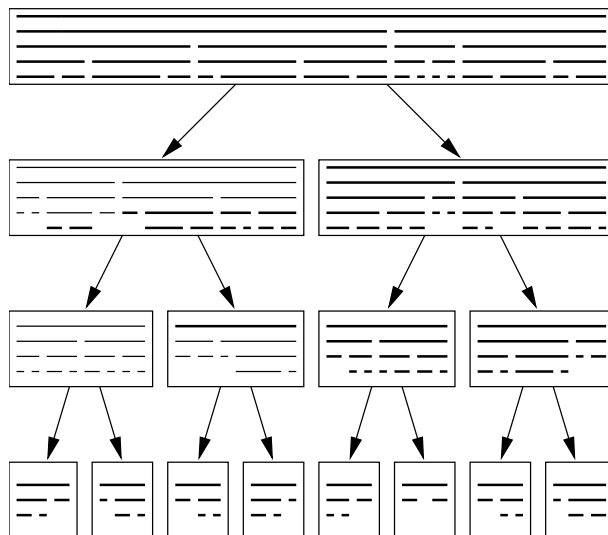
Parallel Delaunay triangulation

Lots of applications, but historically hard to parallelize

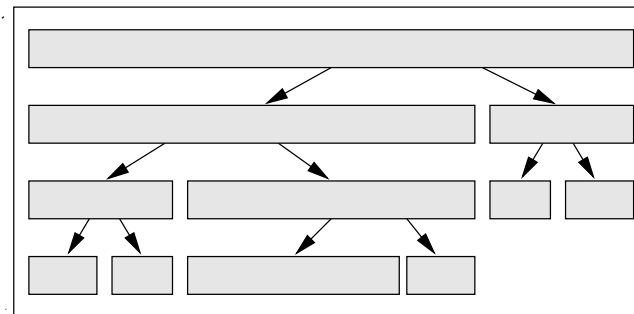
New divide & conquer algorithm by Blelloch, Miller & Talmor ('96)



Outer Delaunay Triangulation



Inner Convex Hull



## How can we implement this?

**HPF** (or other purely data-parallel language)

- Good for regular data structures
- Hard to express control parallelism

**MPI** (or other message-passing library) and e.g. C

- Can express anything, can be efficient
- Hard to get right, very verbose

**NESL** (or other nested data-parallel language)

- Very general and expressive, good for prototyping
- Divide and conquer is subclass of full nested d.p.
- But CVL vector layer is inefficient on RISC MPPs

# My thesis

For *data-parallel divide-and-conquer algorithms*, I can combine NESL's expressibility and MPI's performance.

In particular, I claim that:

- **Data-parallel D&C is an important subclass** of full nested data parallelism.
- **I can devise a D&C programming model** that exploits both control- and data-parallelism.
- **I can automatically generate efficient programs** for distributed-memory machines from this model.

# Summary of results

Speedup over serial code on 32 processor T3D

Speedups are for largest problem that fits on 1 processor (fixed) and for 32 times that size (scaled)

| <i>Algorithm</i>       | <i>Fixed</i> | <i>Scaled</i> |
|------------------------|--------------|---------------|
| Quicksort (worst case) | 10           | 12            |
| Convex hull            | 13           | 24            |
| Geometric separator    | 10           | 14            |
| Delaunay triangulator  | 11           | 17            |

# The rest of the talk

**Examples** of divide-and-conquer algorithms

**Team parallelism:** a new model

**Machiavelli toolkit:** a particular implementation

**Results:** effects of key optimizations

**Related work**

**Conclusions**

# Divide-and-conquer algorithms

Functional structure of a typical D&C algorithm:

```
function D&C ( $p$ )
  if (basecase ( $p$ ))
    return basesolve ( $p$ );
  else
    ( $p_1, p_2, \dots$ ) = divide ( $p$ );
    return combine (D&C ( $p_1$ ),
                    D&C ( $p_2$ ),
                    ...);
```

There is control parallelism in the multiple recursions

But is there data parallelism in **divide** and **combine**?

# Data-parallel D&C algorithms

Sorting:

Quicksort, bitonic sort

Computational geometry:

Closest pairs, convex hull, Delaunay triangulation

Graph theory:

Travelling salesman problem, graph separators

Numerical:

Matrix multiplication, FFT

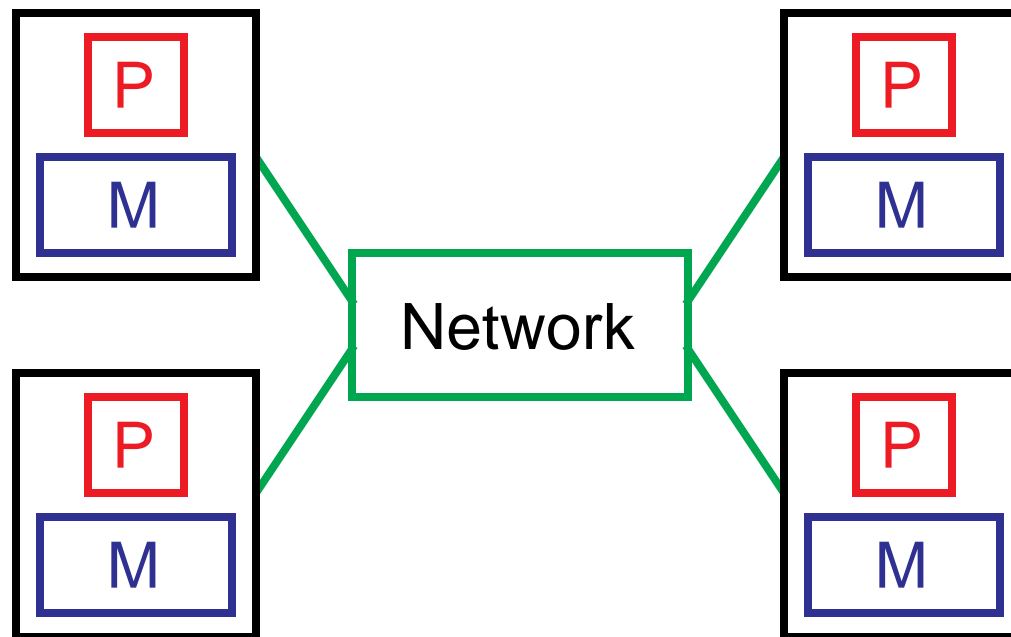
**NOT** data-parallel:

Naive merge sort

# Current machine characteristics

Examples: T3E, SP2, Exemplar, Origin 2000, COW

- Commodity microprocessors
- Physically distributed memory
- High bandwidth network, but high software latency



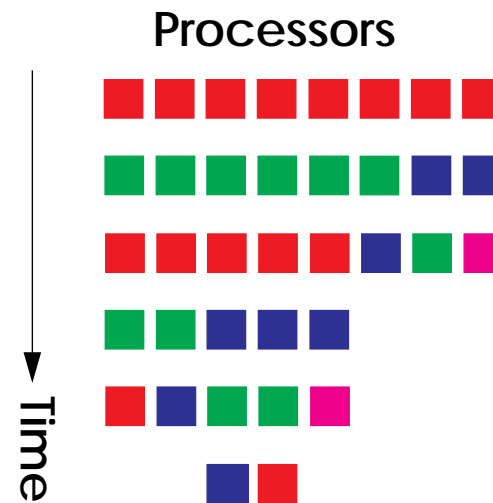
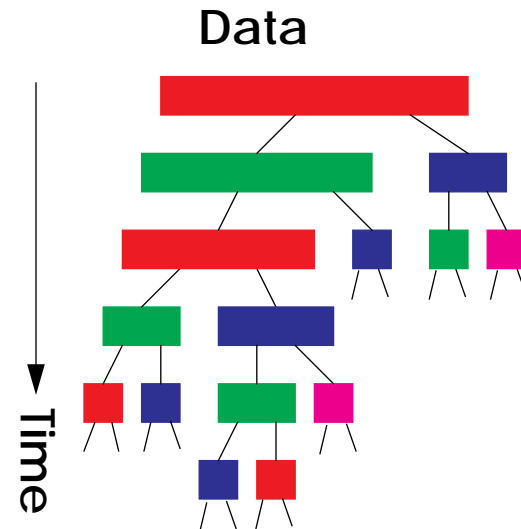
# The team-parallel model

Designed to express  
divide-and-conquer algorithms

SPMD code written in a  
data-parallel style

Four major components:

1. **Data-parallel functions**
2. **Processor teams**
3. **Efficient serial code**
4. **Active load balancing**

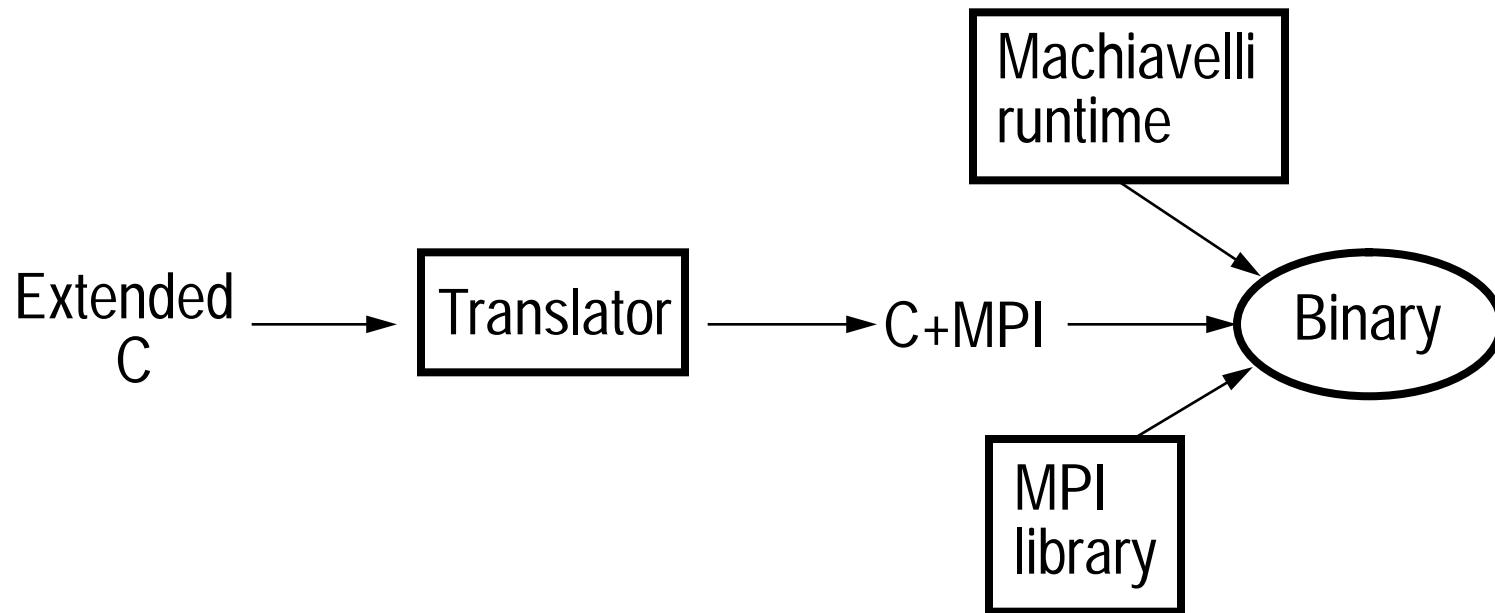


# Machiavelli



An implementation of team parallelism

- Provides vectors as distributed type
- Expressed as a parallel extension to C
- Implemented with source-to-source translator
- Uses MPI for parallelization



# Machiavelli (NESL?) quicksort

```

vec_double quicksort (vec_double s)
{
  vec_double les, eql, grt, l, r, t;
  double pivot;

  if (length (s) < 2) {
    return s;
  } else {
    pivot = get (s, length (s) / 2);
    les = {x : x in s | x < pivot};
    eql = {x : x in s | x == pivot};
    grt = {x : x in s | x > pivot};
    free (s);
    (l, r) = split (quicksort (les),
                  quicksort (grt));
    result = append (l, eql, r);
    free (l); free (eql); free (r);
    return result;
  }
}

```

← basecase (s)  
 ← basesolve (s)

) divide (s)

← combine (s<sub>1</sub>,s<sub>2</sub>)

## Machiavelli extensions to C

A vector can be of any primitive or user-defined type

Basic vector functions:  $O(1)$

**new, free, set, length**

“Local” functions:  $O(n/P)$

Apply-to-each: { **expr : iterators | condition** }

**index, distribute, even, odd, ...**

“Collective” functions:  $O(n/P + k_r \log P + k_s P)$

**get, reduce, scan, append, send, fetch, ...**

Special recursion syntax:

**$(r_0, r_1, \dots) = \text{split} (\text{fn} (\dots), \text{fn} (\dots), \dots)$**

# Implementing split()

```
(left, right) = split (quicksort (les),
                      quicksort (grt));
```

1. *Approximate* cost of calls with user function:

```
int quicksort_cost (s) {
    return s.length * log2 (s.length);
}
```

2. Divide processors into sub-teams accordingly
3. Pack vector arguments (`les`, `grt`) to sub-teams
4. Recurse
5. Spread vector results (`left`, `right`) to old team

Note:  $k$ -way recursion becomes  $k$  teams running singly-recursive code.

## Serial code

Machiavelli generates both parallel and serial code

Serial code uses local operations instead of MPI

User can *override* serial code with a better algorithm:

```
vec_double quicksort_serial (vec_double s) {  
    my_fast_qsort (s.data, s.length);  
    return s;  
}
```

Spend most levels in serial code:  $n \gg P$  so  $\log n \gg \log P$



## An optimization: lazy vectors

Vectors can have unequal amounts of data per proc.

Formed in parallel by apply-to-each with conditionals

```
les = {x : x in s | x < pivot};
eql = {x : x in s | x == pivot};
grt = {x : x in s | x > pivot};
/* les, eql and grt are now unbalanced */
(left, equal) = split (quicksort (les),
                    quicksort (grt));
```

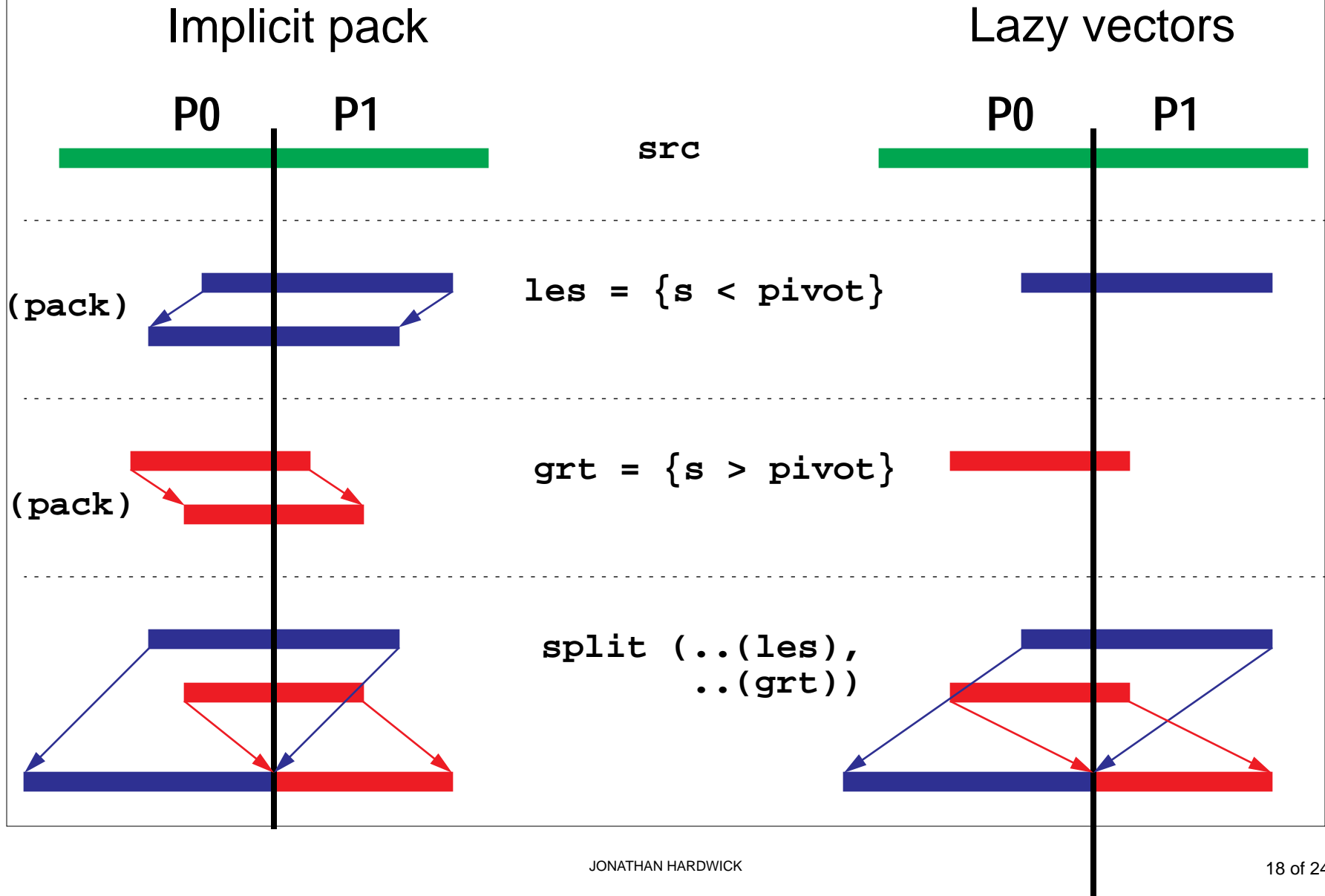
Balancing involves all-to-all communication

But `split()` doesn't care about imbalance!

So mark vectors as *lazy*, balance only when required

- Or when forced by user with special `pack()` function

# Lazy vectors



# Results

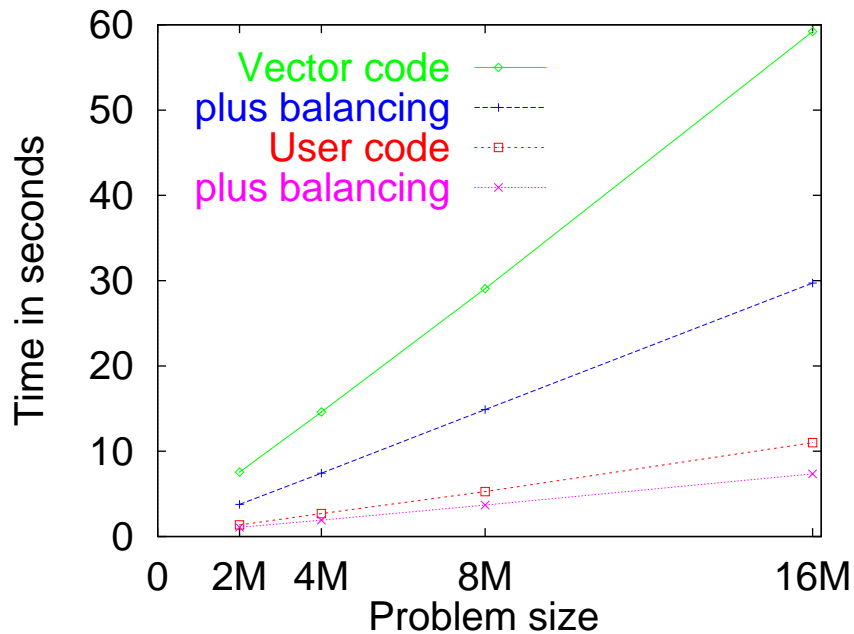
“Men as a whole judge more with their eyes  
than with their hands”

Machiavelli

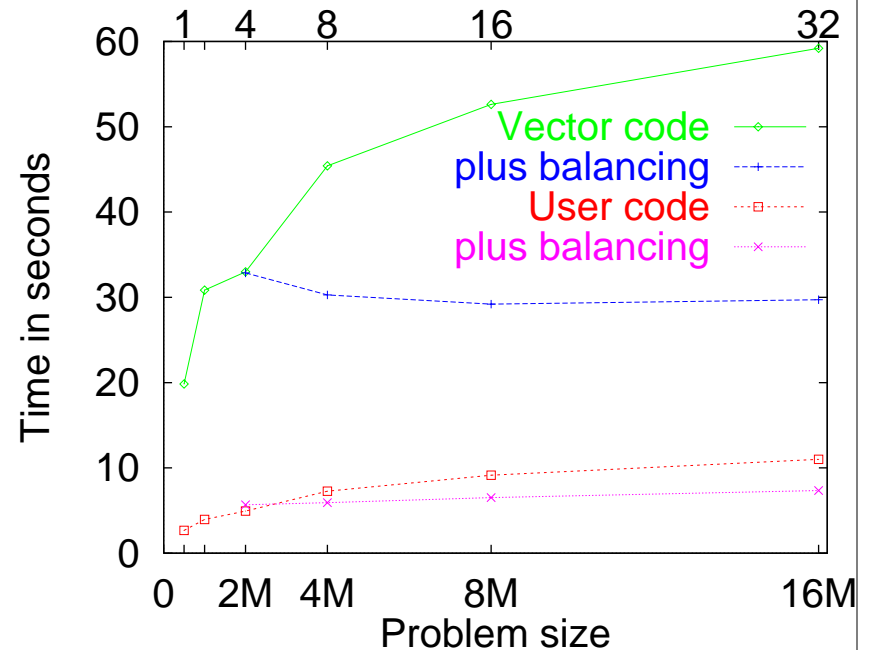
# Effects of load balancing and serial code

Quicksort on T3D (an extreme case):

32 processors



Scaling machine size

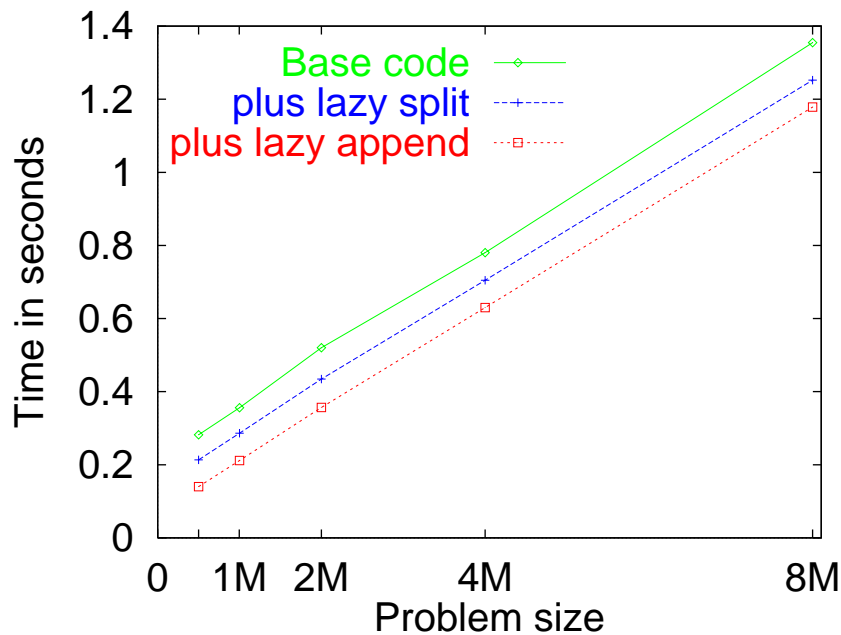


# Effect of lazy vectors

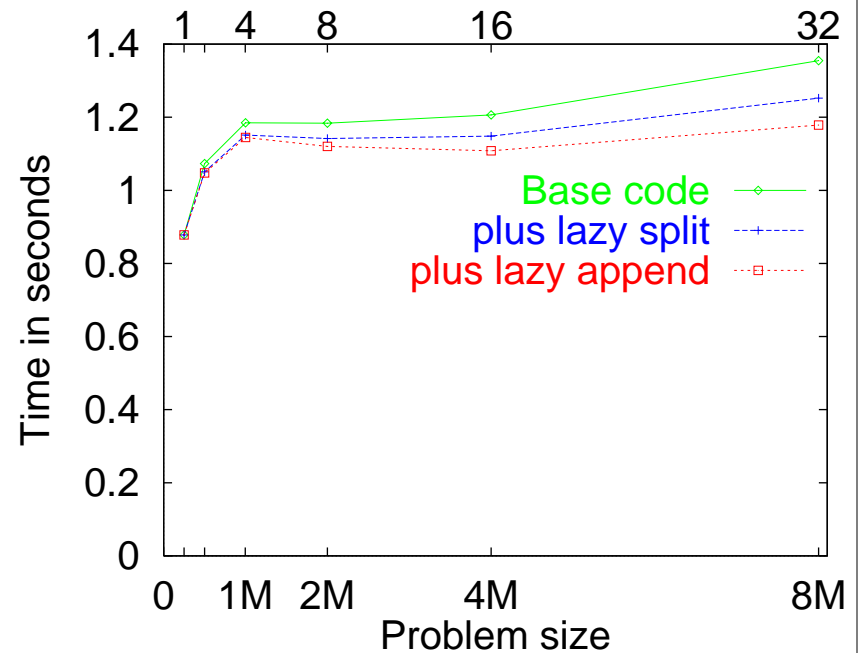
## Convex hull on T3D

Lazy append is extension of lazy split concept, applied after recursion instead of before.

32 processors



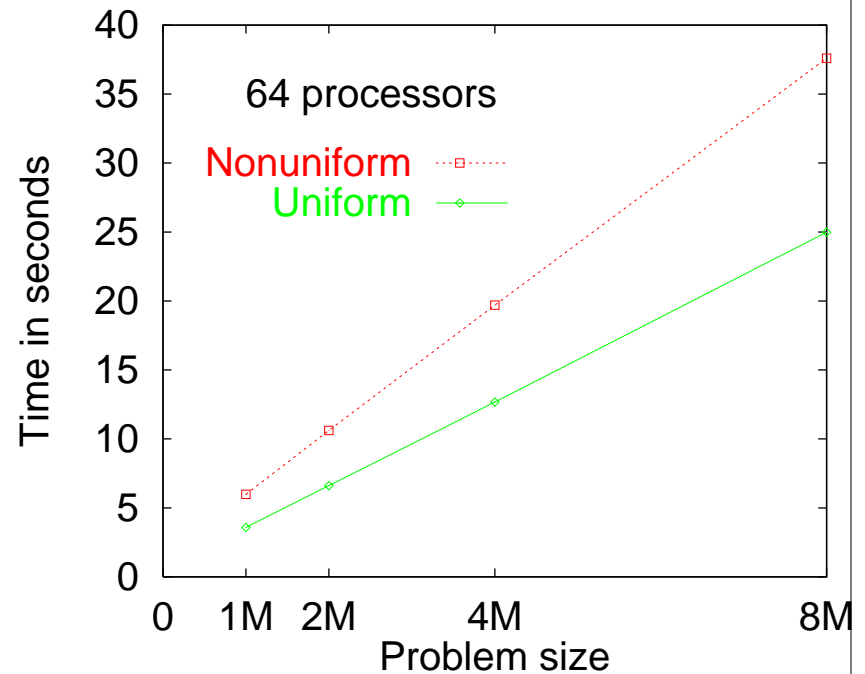
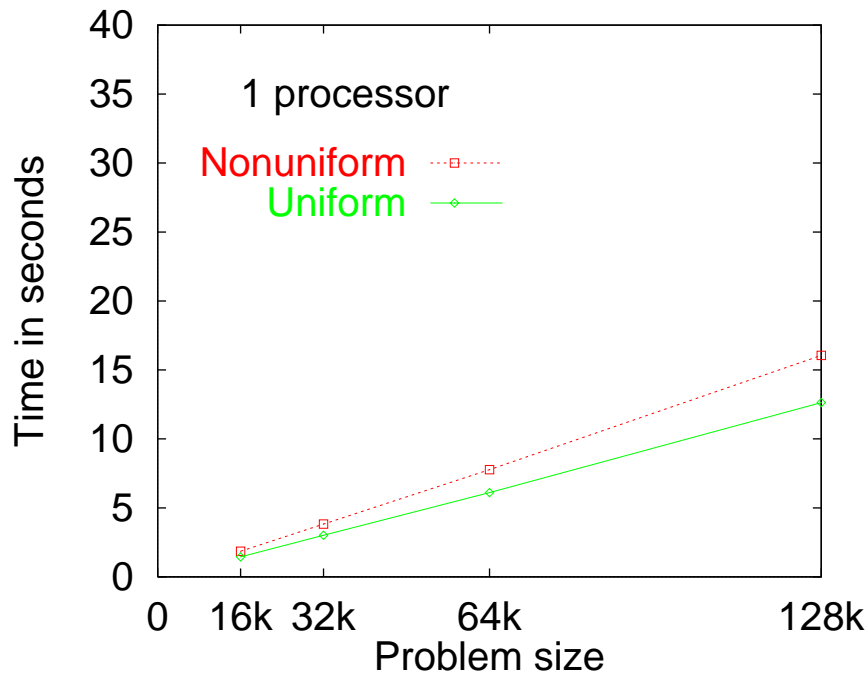
Scaling machine size



# A real application

## Delaunay triangulation on T3D

- Uses Shewchuk’s fast “Triangle” serial code
- Can solve a problem 64 times bigger, on 64 processors, in only about twice the time



## Related work

### Divacon (Mou'90)

- Model of balanced D&C, implemented in \*Lisp

### PCP (Gorda'91)

- D&C language that also has data parallelism
- Requires shared memory, no load balancing

### Concatenated parallelism (Goil'96)

- No data movement in divide - all vectors are lazy
- Can't efficiently implement indexing operations

### Fx (Gross et al'93)

- HPF with concept of static teams
- Now extended with dynamic teams (Subhlok'97)

# Conclusions

I have shown that irregular divide-and-conquer algorithms can be implemented efficiently on distributed-memory machines. I have also:

Defined team parallelism

- SPMD data-parallel model for D&C algorithms

Developed Machiavelli

- A particular implementation of team parallelism

Produced fast parallel algorithms

- e.g., world's fastest Delaunay triangulator

Categorized parallel D&C algorithms

- (In the dissertation)