Fork/Join Parallelism in Java

Doug Lea

State University of New York at Oswego

`dl@cs.oswego.edu`

`http://gee.cs.oswego.edu`

Outline

Fork/Join Parallel Decomposition

A Fork/Join Framework

Recursive Fork/Join programming

Empirical Results

Parallel Decomposition

Goal: Minimize service times by exploiting parallelism

Approach:

Partition into subproblems

Break up main problem into several parts. Each part should be as independent as possible.

Create subtasks

Construct each solution to each part as a `Runnable` task.

Fork subtasks

Feed subtasks to pool of worker threads. Base pool size on number of CPUs or other resource considerations.

Join subtasks

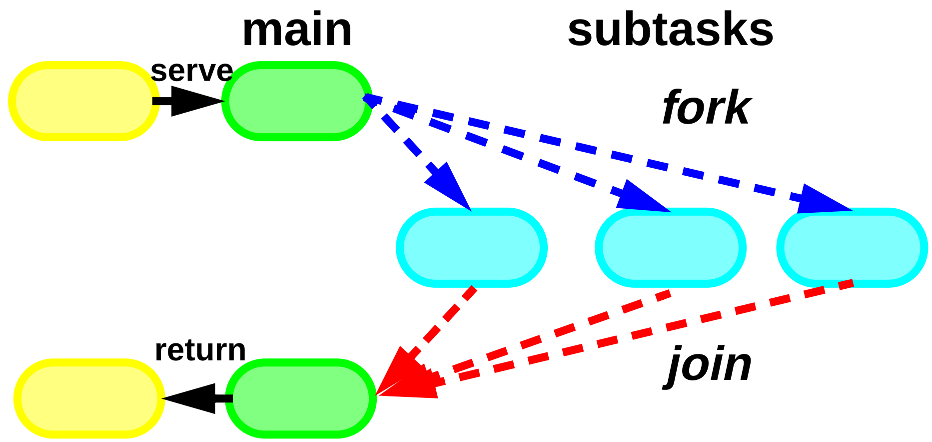
Wait out processing of as many subtasks (usually all) needed to compose solution

Compose solution

Compose overall solution from completed partial solutions. (aka *reduction*, *agglomeration)*

Fork/Join Parallelism

Main task must help synchronize and schedule subtasks




```
public Result serve(Problem problem) {
  SPLIT the problem into parts;

  FORK:
    for each part p
      create and start task to process p;

  JOIN:
    for each task t
      wait for t to complete;

  COMPOSE and return aggregate result;
}
```

Task Granularity

How big should each task be?

- Approaches and answers differ for different kinds of tasks
  - Computation-intensive, I/O-intensive, Event-intensive
- Focus here on computation-intensive

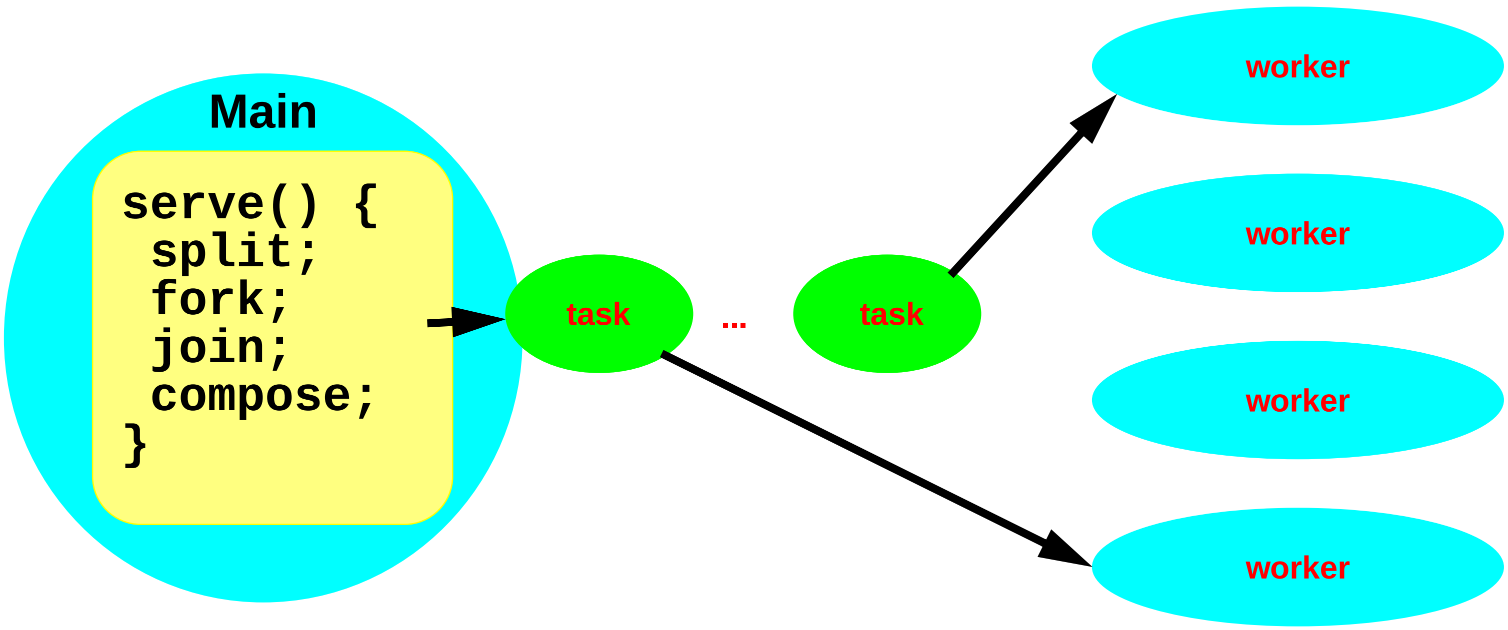
Two opposing forces:

- To maximize parallelism, make each task as small as possible
  - Improves load-balancing, locality, decreases percentage of time that CPUs idly wait for each other, and leads to greater throughput
- To minimize overhead, make each task as large as possible
  - Creating, enqueing, dequeing, executing, maintaining status, waiting for, and reclaiming resources for Task objects add overhead compared to direct method calls.

Must adopt an engineering compromise:

- Use special-purpose low-overhead Task frameworks
- Use parameterizable decomposition methods that rely on sequential algorithms for small problem sizes

Fork/Join with Worker Threads



Each worker thread runs many tasks

- Java Threads are too heavy for direct use here.

Further opportunities to improve performance

- Exploit simple scheduling properties of fork/join
- Exploit simple structure of decomposed tasks

Simple Worker Threads

Establish a producer-consumer chain

- Producer
  - Service method just places task in a channel
    - Channel might be a buffer, queue, stream, etc
    - Task might be represented by a `Runnable` command, event, etc
- Consumer
  - Host contains an autonomous loop thread of form:

```
while (!Thread.interrupted()) {
  task = channel.take();
  process(task);
}
```

Worker Thread Example

```
interface Channel { // buffer, queue, stream, etc
  void   put(Object x);
  Object take();
}

class Host { //...
  Channel channel = ...;
  public void serve(...) {
    channel.put(new Runnable() { // enqueue
      public void run(){
        handler.process(...);
      }});
  }

  Host() {   // Set up worker thread in constructor
    // ...
    new Thread(new Runnable() {
      public void run() {
       while (!Thread.interrupted())
        ((Runnable)(channel.take())).run();
      }
    }).start();
  }
}
```

A Task Framework

Fork/Join Task objects can be much lighter than `Thread` objects

- No blocking except to join subtasks
  - Tasks just run to completion
  - Cannot enforce automatically, and short-duration blocking is OK anyway.
- Only internal bookkeeping is completion status bit.
- All other methods relay to current worker thread.

```
abstract class FJTask implements Runnable {
  boolean isDone();       // True after task is run
  void fork();            // Start a dependent task
  static void yield(); // Allow another task to run
  void join();                   // Yield until isDone
  static void invoke(Task t);    // Directly run t
  static void coInvoke(Task t,Task u);// Fork+join
  static void coInvoke(Task[] v); // Fork+join all
  void reset();                        // Clear isDone
  void cancel();                       // Force isDone
} // (plus a few others)
```

Fork/Join Worker Thread Pools

Uses per-thread queuing with work-stealing

- **Normally best to have one worker thread per CPU**
  - **But design is robust. It scarcely hurts (and sometimes scarcely helps) to have more workers than CPUs**
- **Each new task is queued in current worker thread's dequeue (double-ended queue)**
  - **Plus a global entry queue for new tasks from clients**
- **Workers run tasks from their own dequeues in stack-based LIFO (i.e., newest task first) order.**
- **If a worker is idle, it steals a task, in FIFO (oldest task first) order from another thread's dequeue or entry queue**

Work-Stealing

Original algorithm devised in Cilk project (MIT)

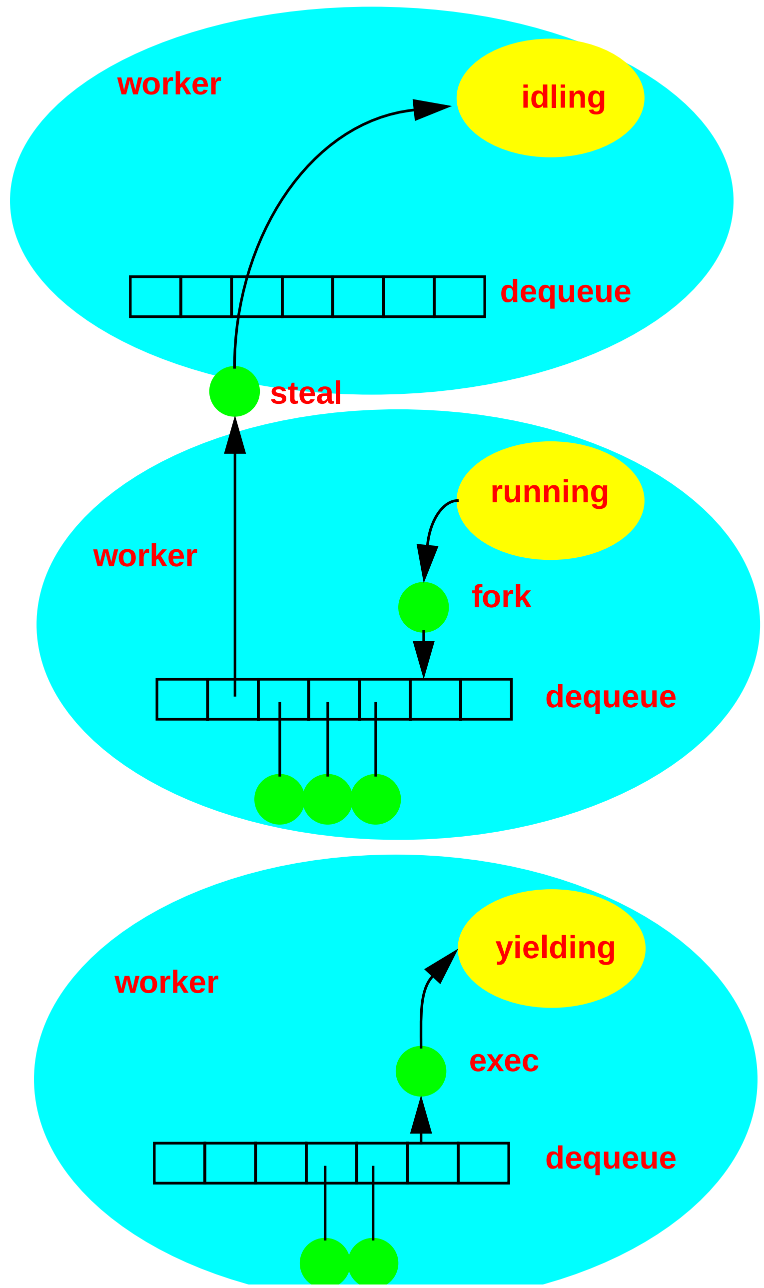
- Several variants
- Shown to scale on stock MP hardware

Leads to very portable application code

Typically, the only platform-dependent parameters are:

- Number of worker threads
- Problem threshold size for using sequential solution

Works best with recursive decomposition



Recursive Decomposition

Typical algorithm:

```
Result solve(Param problem) {
  if (problem.size <= GRANULARITY_THRESHOLD)
    return directlySolve(problem);
  else {
     in-parallel {
       Result l = solve(lefthalf(problem));
       Result r = solve(rightHalf(problem);
     }
     return combine(l, r);
  }
}
```

Why?

Support tunable granularity thresholds

Under work-stealing, the algorithm itself drives the scheduling

There are known recursive decomposition algorithms for many computationally-intensive problems.

Some are explicitly parallel, others are easy to parallelize

Example: Fibonacci

A useless algorithm, but easy to explain!

Sequential version:

```
int seqFib(int n) {
  if (n <= 1)
    return n;
  else
    return seqFib(n-1) + seqFib(n-2);
}
```

To parallelize:

- **Replace function with Task subclass**
  - **Hold arguments/results as instance vars**
  - **Define `run()` method to do the computation**
- **Replace recursive calls with fork/join Task mechanics**
  - **`Task.coinvoke` is convenient here**
- **But rely on sequential version for small values of n**

  Threshold value usually an empirical tuning constant

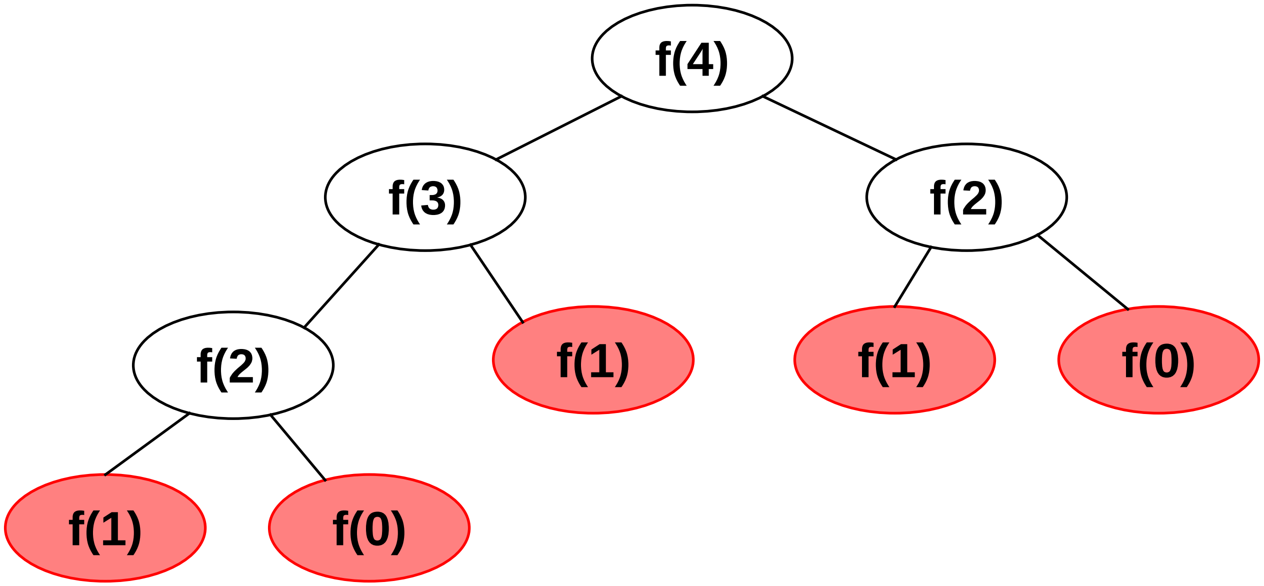
Class Fib

```
class Fib extends FJTask {
  volatile int number; // serves as arg and result
  Fib(int n) { number = n; }

  public void run() {
    int n = number;
    if (n <= 1) { /* do nothing */ }
    else if (n <= sequentialThreshold) //(12 works)
      number = seqFib(n);
    else {
      Fib f1 = new Fib(n - 1);         // split
      Fib f2 = new Fib(n - 2);
      coInvoke(f1, f2);                // fork+join
      number = f1.number + f2.number; // compose
    }
  }

  int getAnswer() { // call from external clients
    if (!isDone())
      throw new Error("Not yet computed");
    return number;
  }
}
```

Fib Server

```
public class FibServer  { // Yes. Very silly
  public static void main(String[] args) {
    TaskRunnerGroup group = new
      TaskRunnerGroup(Integer.parseInt(args[0]));
    ServerSocket socket = new ServerSocket(1618);
    for (;;) {
      final Socket s = socket.accept();
      group.execute(new Task() {
        public void run() {
          DataInputStream i = new
            DataInputStream(s.getInputStream());
          DataOutputStream o = new
            DataOutputStream(s.getOutputStream());
          Fib f = new Fib(i.readInt());
          invoke(f);
          o.writeInt(f.getAnswer());
          s.close()
        });
      }
    }
  }
} // (Lots of exception handling elided out)
```

Computation Trees



Recursive computation meshes well with work-stealing:

- **With only one worker thread, computation proceeds in same order as sequential version**
  - **The local LIFO rule is same as, and not much slower than recursive procedure calls**
- **With multiple threads, other workers will typically steal larger, non-leaf subtasks, which will keep them busy for a while without further inter-thread interaction**

Iterative Computation

Many computation-intensive algorithms have structure:

Break up problem into a set of tasks, each of form:

- **For a fixed number of steps, or until convergence, do:**
  - **Update one section of a problem;**
  - **Wait for other tasks to finish updating their sections;**

Examples include mesh algorithms, relaxation, physical simulation

Illustrate with simple Jacobi iteration, with base step:

```
void oneStep(double[][] oldM, double[][] newM,
             int i, int j) {
   newM[i][j] = 0.25 * (oldM[i-1][j] +
                        oldM[i][j-1] +
                        oldM[i+1][j] +
                        oldM[i][j+1]);
}
```

Where `oldM` and `newM` alternate across steps

Iteration via Computation Trees

Explicit trees avoid repeated problem-splitting across iterations

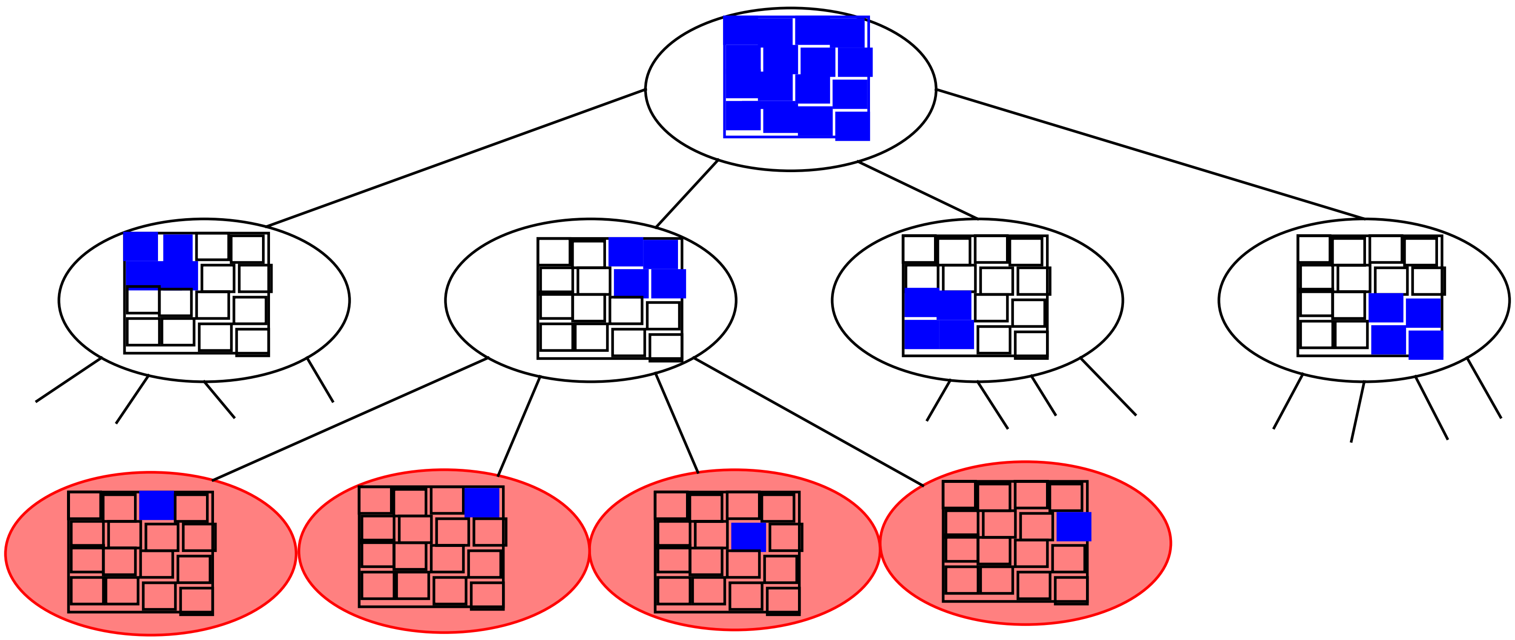
Allow Fork/Join to be used instead of barrier algorithms

For Jacobi, can recursively divide by quadrants

- **Leaf nodes do computation;**

  Leaf node size (cell count) is granularity parameter

- **Interior nodes drive task processing and synchronization**



Jacobi example

```
abstract class Tree extends Task {
  volatile double maxDiff; //for convergence check
}


class Interior extends Tree {
  final Tree[] quads;

  Interior(Tree q1, Tree q2, Tree q3, Tree q4) {
    quads = new Tree[] { q1, q2, q3, q4 };
  }

  public void run() {
    coInvoke(quads);
    double md = 0.0;
    for (int i = 0; i < 4; ++i) {
      md = Math.max(md,quads[i].maxDiff);
      quads[i].reset();
    }
    maxDiff = md;
  }
}
```

Leaf Nodes

```
class Leaf extends Tree {
  final double[][] A; final double[][] B;
  final int loRow; final int hiRow;
  final int loCol; final int hiCol; int steps = 0;
  Leaf(double[][] A, double[][] B,
       int loRow, int hiRow,
       int loCol, int hiCol) {
    this.A = A;   this.B = B;
    this.loRow = loRow; this.hiRow = hiRow;
    this.loCol = loCol; this.hiCol = hiCol;
  }
  public synchronized void run() {
    boolean AtoB = (steps++ % 2) == 0;
    double[][] a = (AtoB)? A : B;
    double[][] b = (AtoB)? B : A;
    for (int i = loRow; i <= hiRow; ++i) {
      for (int j = loCol; j <= hiCol; ++j) {
        b[i][j] = 0.25 * (a[i-1][j] + a[i][j-1] +
                          a[i+1][j] + a[i][j+1]);
        double diff = Math.abs(b[i][j] - a[i][j]);
        maxDiff = Math.max(maxDiff, diff);
      }
    }
} }
```

Driver

```
class Driver extends Task {
  final Tree root;  final int maxSteps;
  Driver(double[][] A, double[][] B,
         int firstRow, int lastRow,
         int firstCol, int lastCol,
         int maxSteps, int leafCells) {
    this.maxSteps = maxSteps;
    root = buildTree(/* ... */);
  }

  Tree buildTree(/* ... */) { /* ... */}

  public void run() {
    for (int i = 0; i < maxSteps; ++i) {
      invoke(root);
      if (root.maxDiff < EPSILON) {
        System.out.println("Converged");
        return;
      }
      else
        root.reset();
    }
  }
}
```

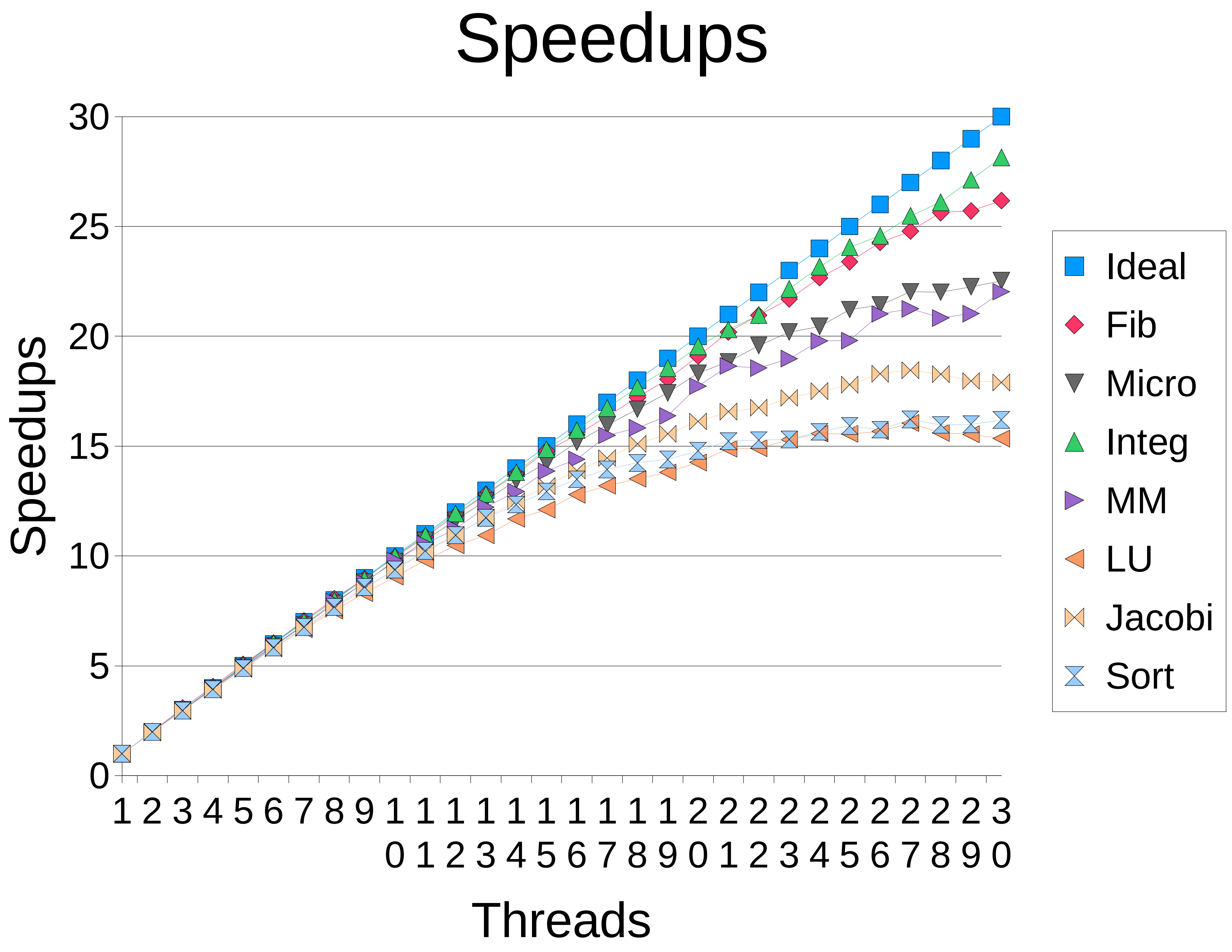
Performance

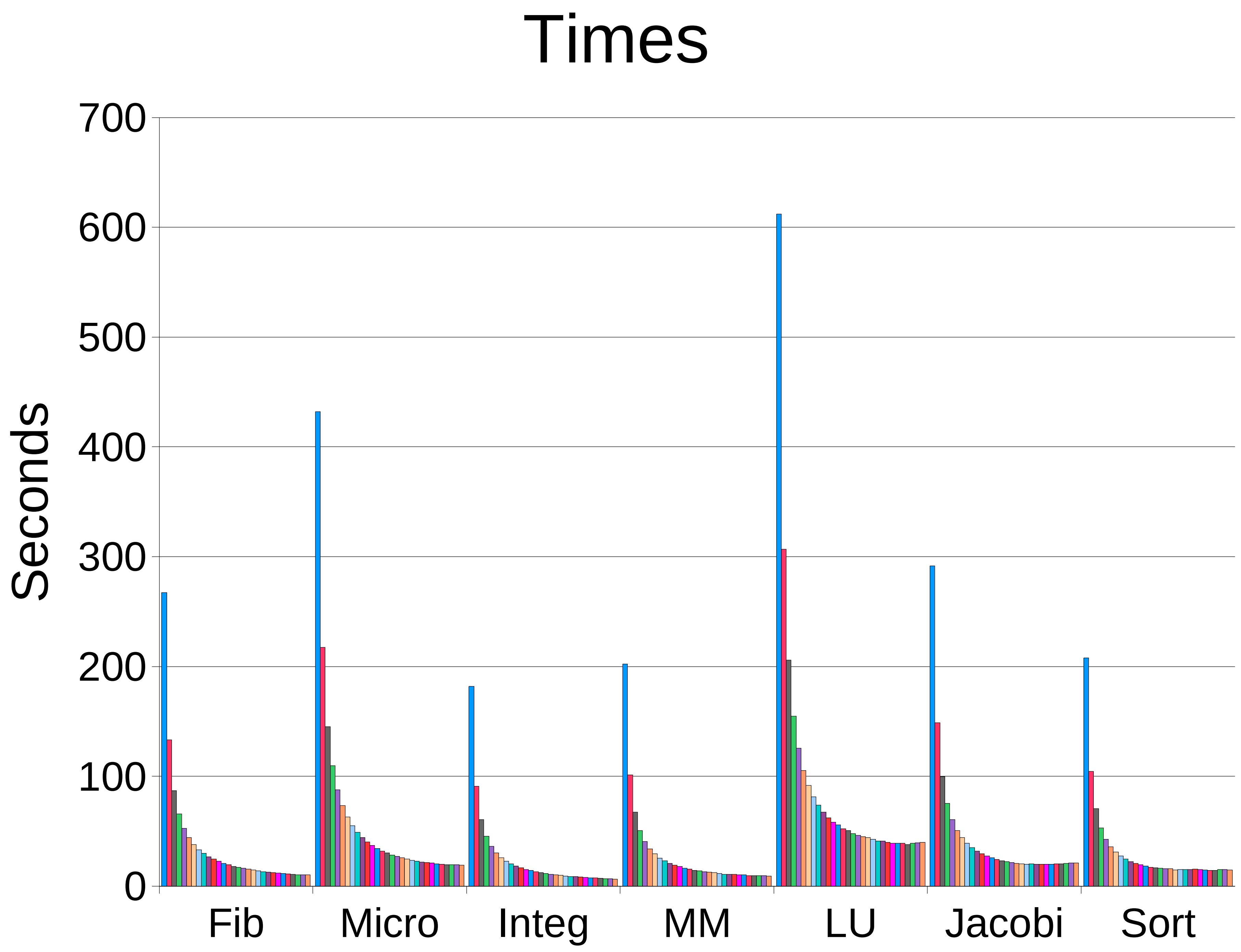
Test programs

- **Fib**
- **Matrix multiplication**
- **Integration**
- **Best-move finder for game**
- **LU decomposition**
- **Jacobi**
- **Sorting**

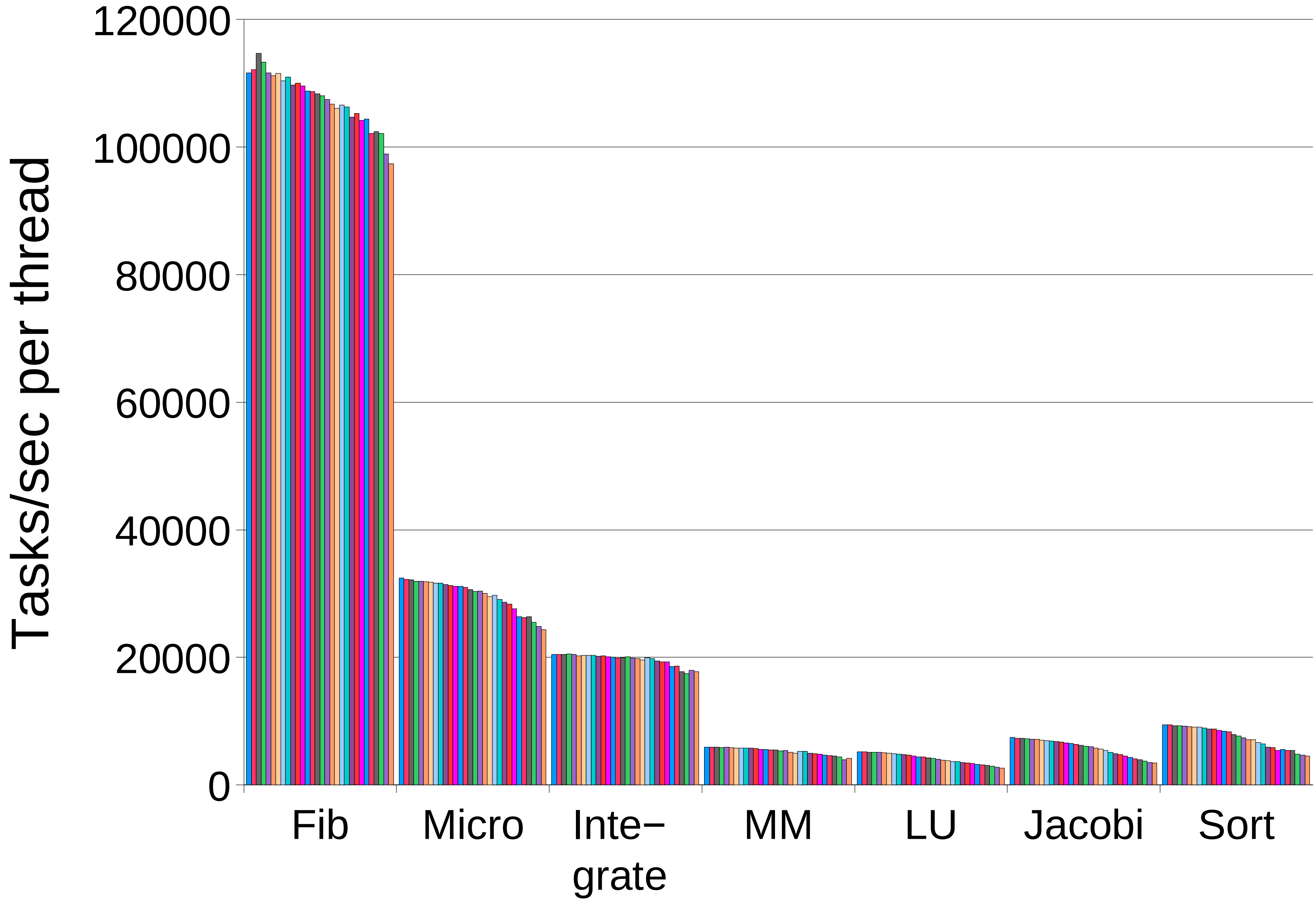
Main test platform

- **30-CPU Sun Enterprise**
- **Solaris Production 1.2.x JVM**

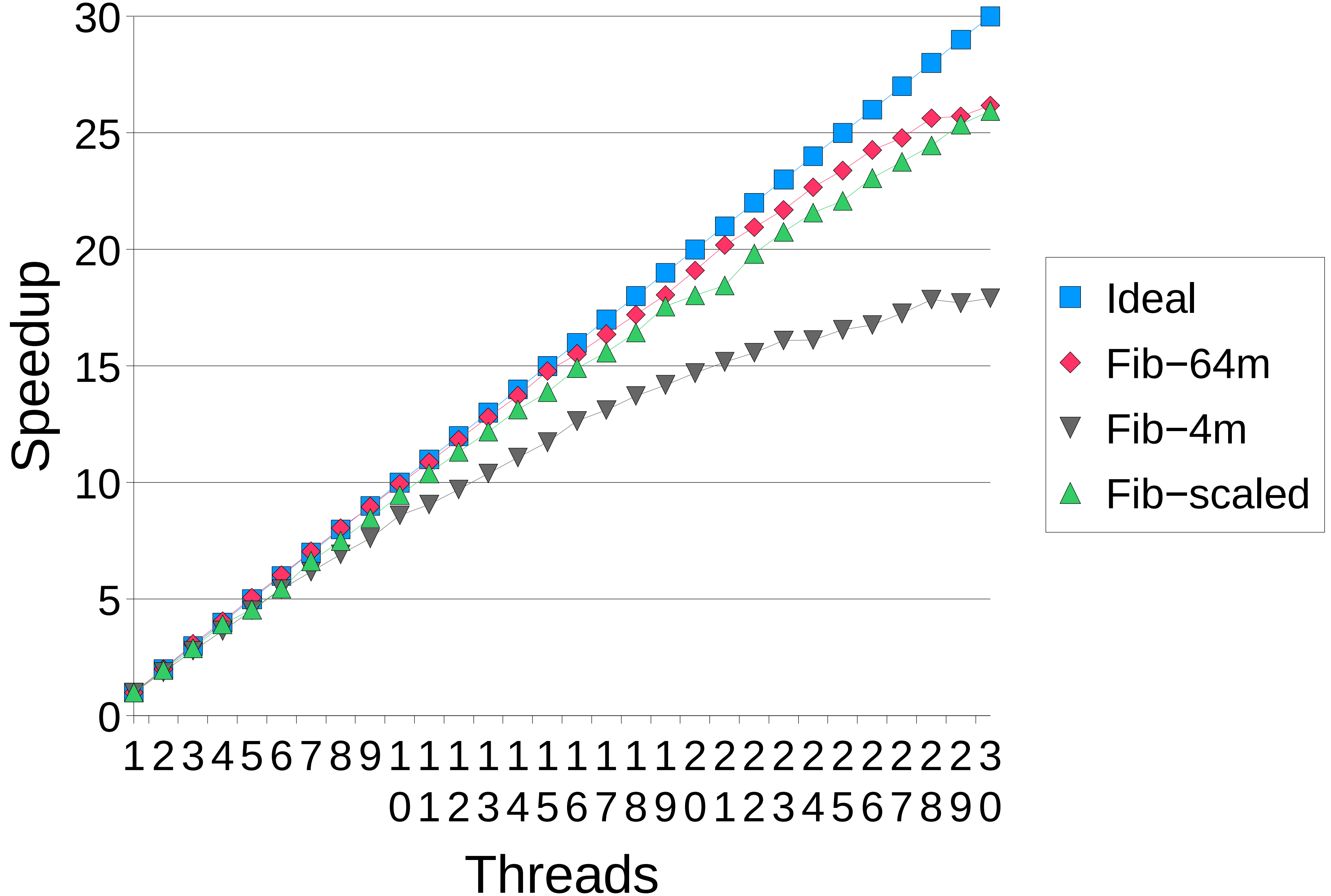

Speedups
Speedups
Threads
0
5
10
15
20
25
30
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
Ideal
Fib
Micro
Integ
MM
LU
Jacobi
Sort


Times
Seconds
700
600
500
400
300
200
100
0
Fib
Micro
Integ
MM
LU
Jacobi
Sort

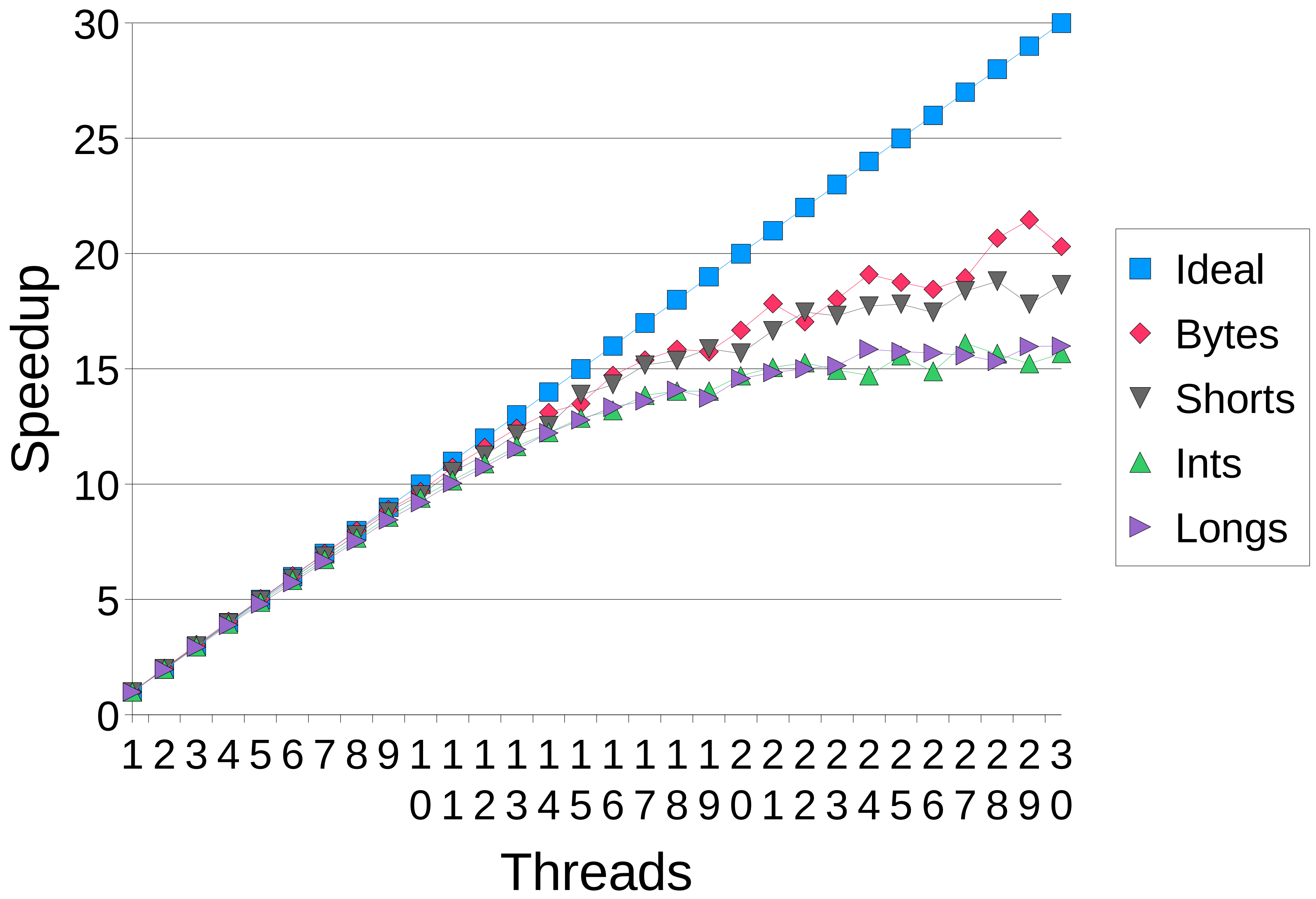
Task rates


Tasks/sec per thread
120000
100000
80000
60000
40000
20000
0
Fib
Micro
Inte−
grate
MM
LU
Jacobi
Sort

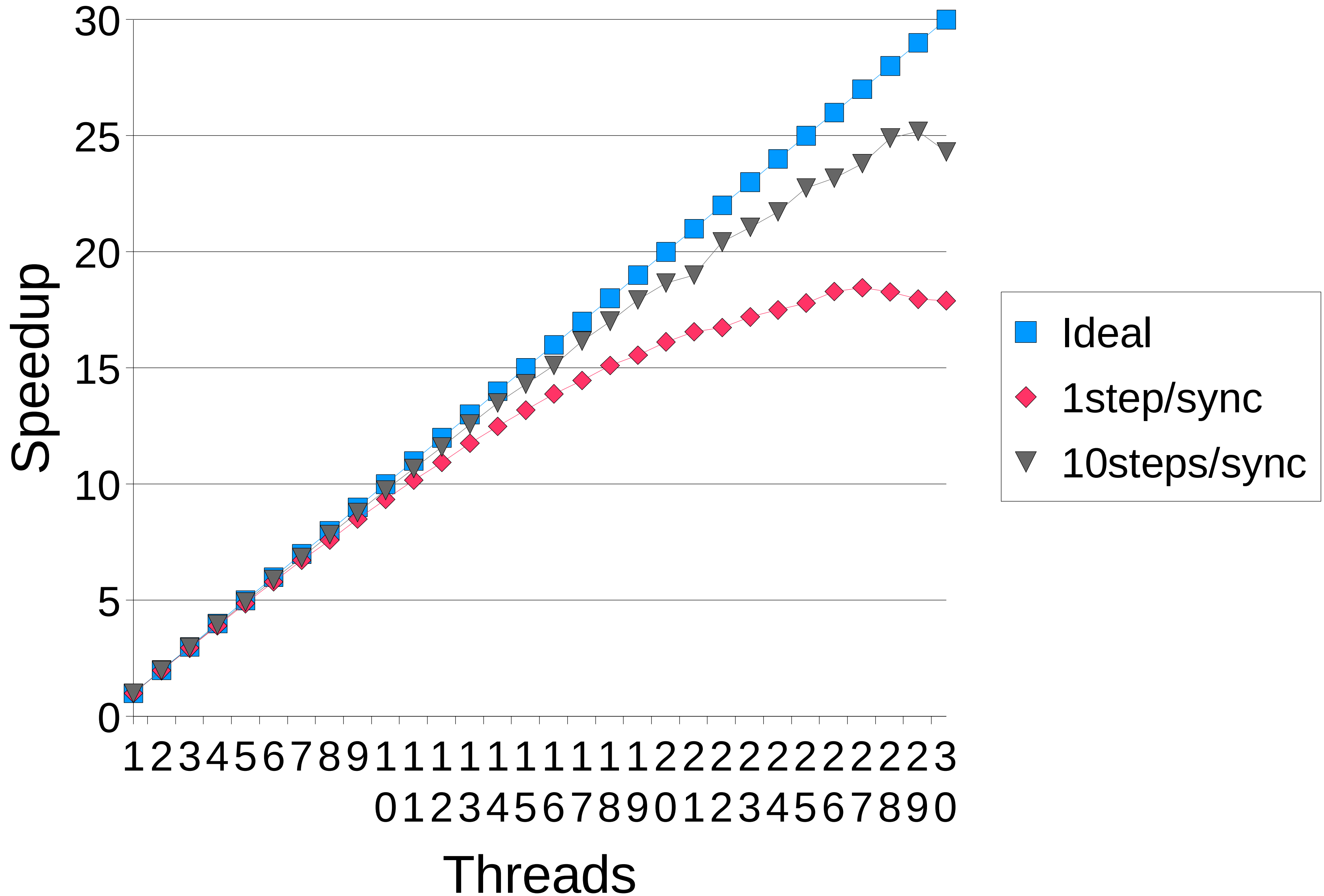
GC Effects: FIb

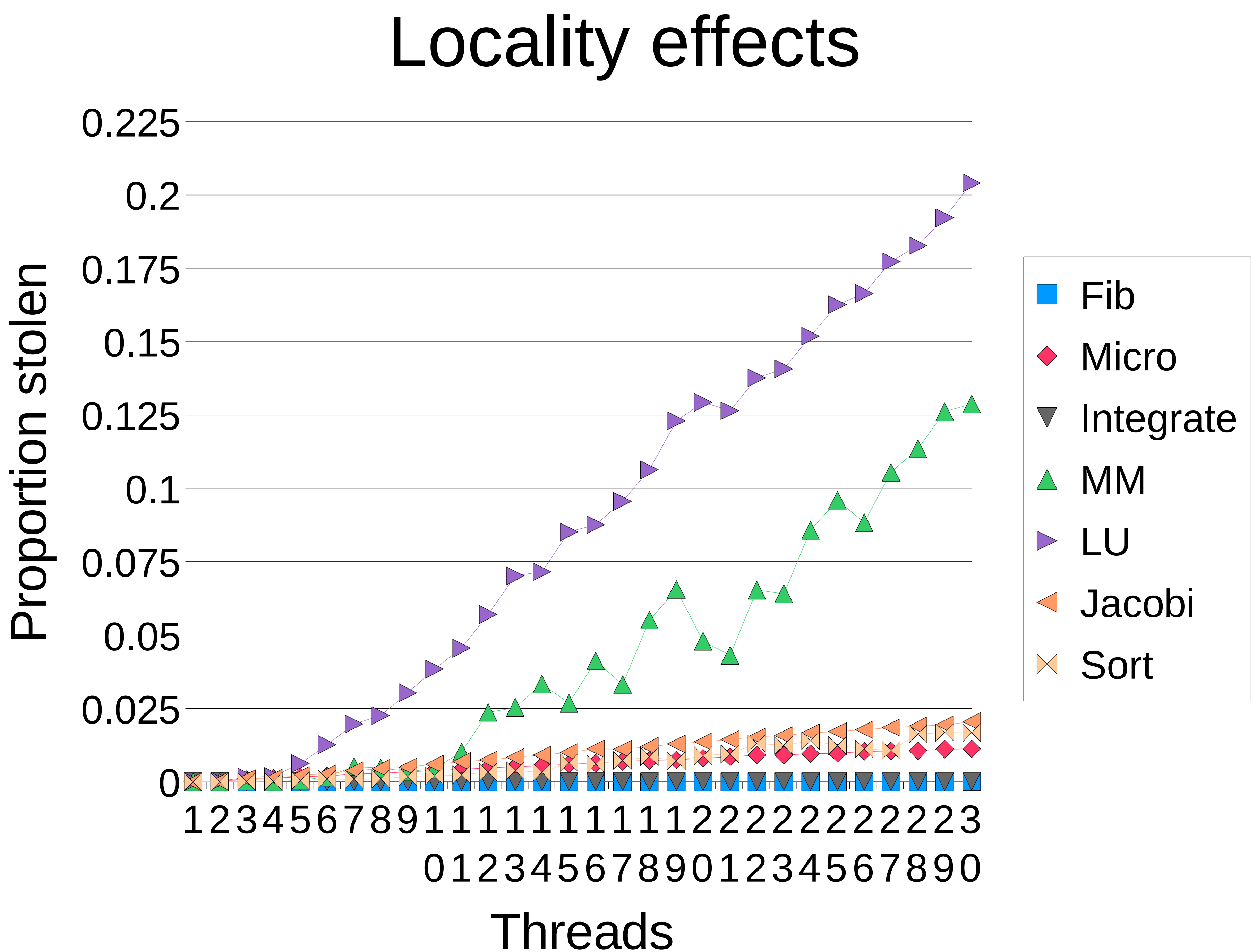


Memory bandwidth effects: Sorting



Sync Effects: Jacobi




Locality effects
Proportion stolen
0.225
0.2
0.175
0.15
0.125
0.1
0.075
0.05
0.025
0
Threads
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
Fib
Micro
Integrate
MM
LU
Jacobi
Sort

Other Frameworks