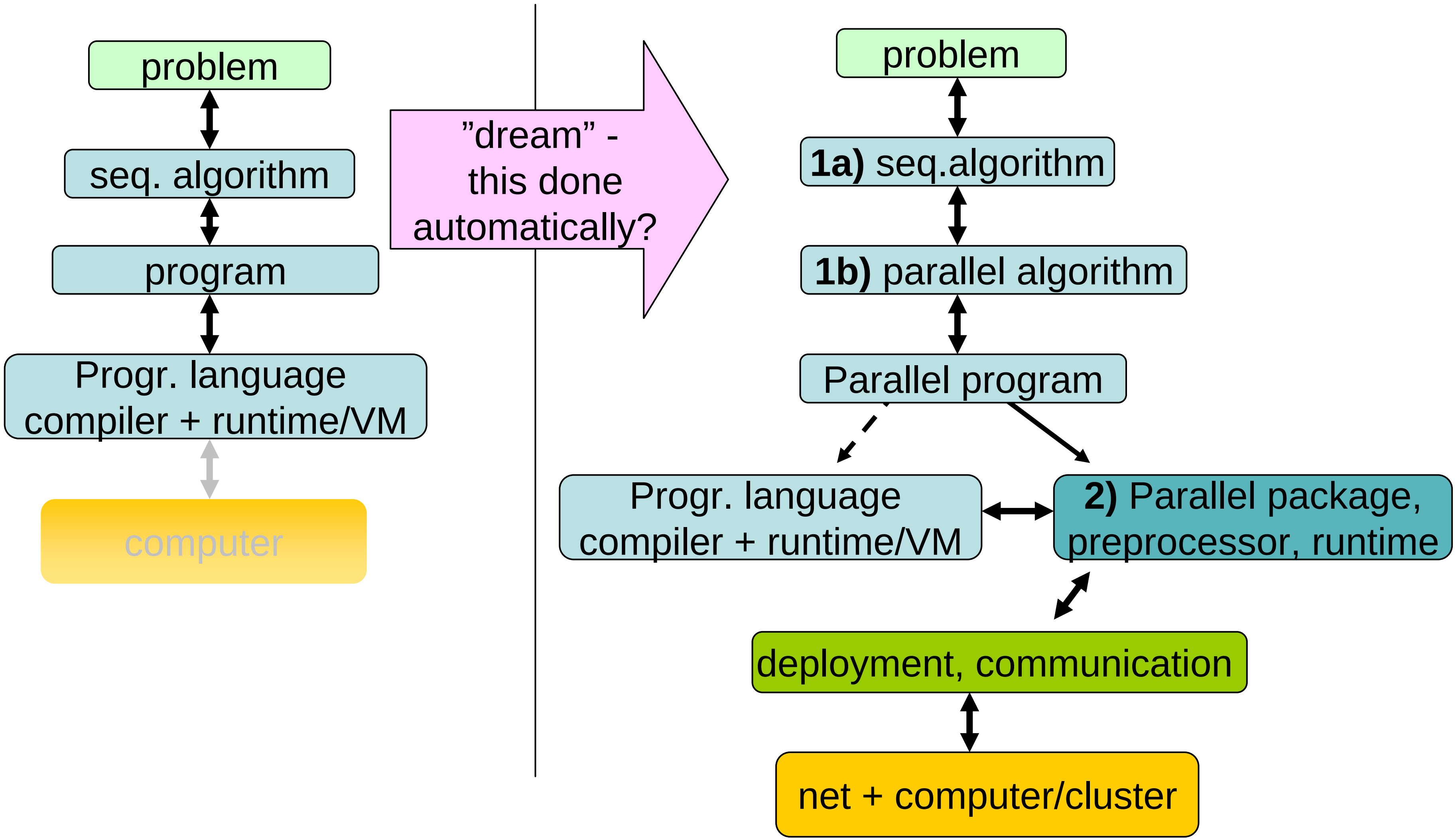
Automatic parallelization and Parallel Recursive Procedures (PRP) – a parallel package and its implementation

Arne Maus,
with **Torfinn Aas, Yan Xu, Arne Høstmark, Viktor Eide, Tore André Rønningen, André Næss, Mads Bue, Christian O. Søhoel, Bjørn Arild Kristiansen, Coung Van Truong, Jørn Christian Syvertsrud**
Dept. of Informatics,
Univ. Oslo

Overview

1. Problems, promises and limitations in parallel programming.
2. PRP - a programming package for ‘automatic’ parallel programming
3. A few results
4. How to implement PRP

sequential vs. parallel program

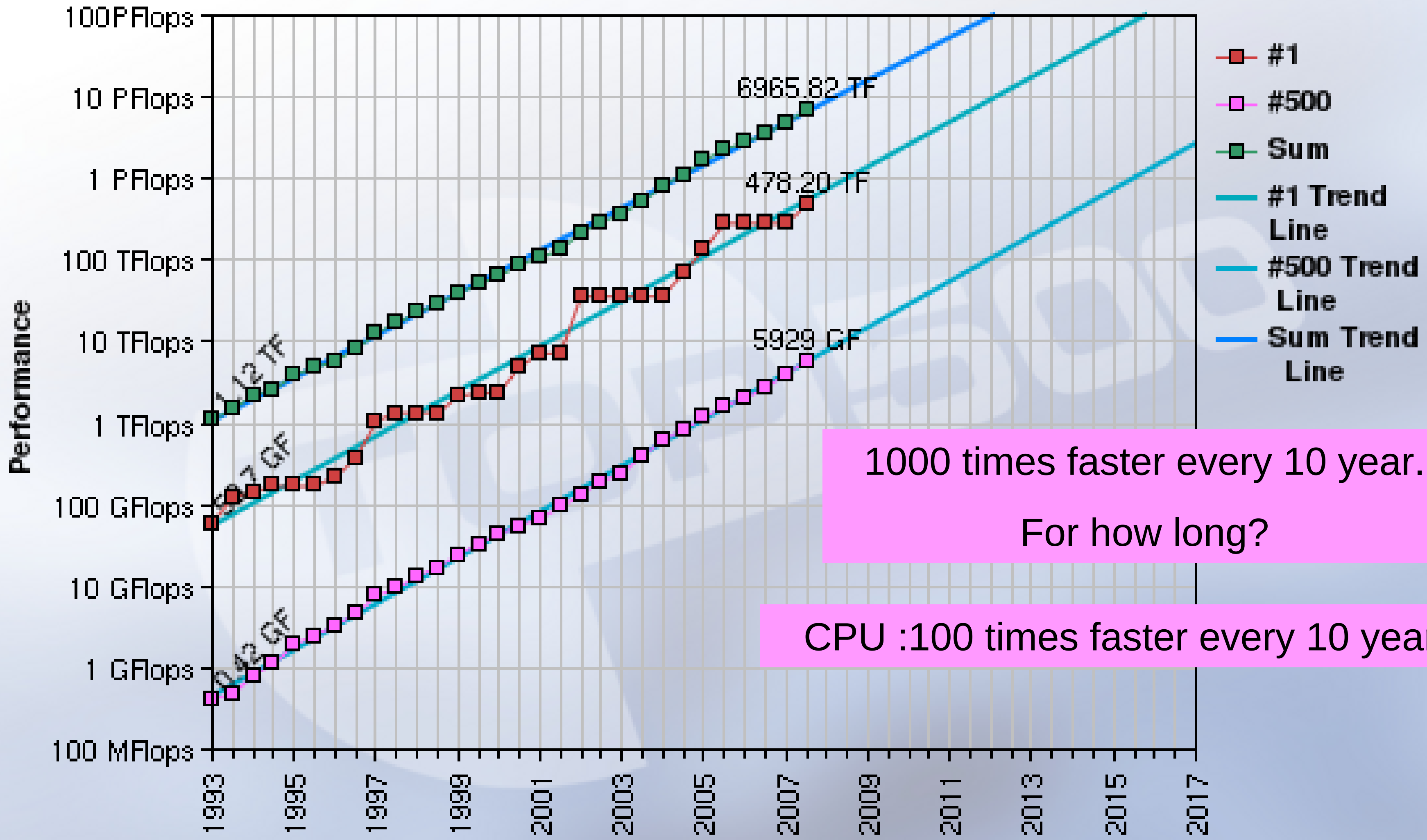


1. Limitations of Parallel programming

- **How fast a computer can you ever build, how large a problem can ever be solved with PP ?**
  - Today's fastest (top 500)
  - How many instruction can ever be performed (in all history) ?
  - , and in practice
  - In your office ‘soon’
- **Can all algorithms be parallelized?**
  - How much speedup: Amdahl's law
  - Can all problems be parallelized:
    - 20% (yes) + 60% (some parts) + 20 % (no)
  - How to transform a sequential to a parallel algorithm
- **Matching the algorithm to the CPU/GPU/ computer/cluster**
    - Little or no shared memory
    - Net delay
    - load balancing
    - fault tolerance

TOP500 SUPERCOMPUTER SITES

Projected Performance Development



1000 times faster every 10 year.

For how long?

CPU :100 times faster every 10 year.

TOP500 List - November 2007 (1-100)

R_{max} and R_{peak} values are in GFlops. For more details about other fields, check the TOP500 description.

nex

Rank	Site	Computer	Processors	Year	R_{max}	R_{peak}
1	DOE/NNSA/LLNL United States	BlueGene/L - eServer Blue Gene Solution IBM	212992	2007	478200	596378
2	Forschungszentrum Juelich (FZJ) Germany	JUGENE - Blue Gene/P Solution IBM	65536	2007	167300	222822
3	SGI/New Mexico Computing Applications Center (NMCAC) United States	SGI Altix ICE 8200, Xeon quad core 3.0 GHz SGI	14336	2007	126900	172032
4	Computational Research Laboratories, TATA SONS India	EKA - Cluster Platform 3000 BL460c, Xeon 53xx 3GHz, Infiniband Hewlett-Packard	14240	2007	117900	170880
5	Government Agency Sweden	Cluster Platform 3000 BL460c, Xeon 53xx 2.66GHz, Infiniband Hewlett-Packard	13728	2007	102800	146430

How many instruction can ever be performed?

- **Assume that each elementary particle in the universe is turned into a CPU:**
  - connected as a perfect parallel machine :**10^{80}** CPUs
  - cycle time = the time it takes light to pass an atom nucleus: $3*10^6$ km/sec / 10^{-15} m = **$3 * 10^{24}$ Hz**
  - duration of computation: The Earth is destroyed by the Sun in 5 billion years = 60*60*24*365*5000 000 000 sec. $\leq$ 1,57 * **10^{16} sec**
- **In total $\leq$ 10^{121} operations at 10^{105} flops**

Easy to construct ***problems larger*** than **10^{121}** operations, i.e. the 100 queens problem, or The Travelling Salesman (using the naive recursive decent algorithm) for 100 cities.

More 'realistic' assumptions:

a) **1000 ton of CPUs each 0.1 gram at 1000 GHz gives** **10^{24}** flops

b) **PC with 10^5-10^6 multi-core CPU**
(line width = 0.5 - 0.1 nm = 50-10 atoms) at your desktop,
each at 10 -100 GHz gives:

10^{15}- 10^{17} flops or 1 - 100 Petaflops

And if you will only wait for a week ($\leq 10^6$ sec.), then anything more than 10^{20} - 10^{30} operations for solving a problem, are unrealistic

Amdahl's law and a conclusion

- Amdahl:

  If the problem has a fixed sequential part of p %, then **100/p** is the maximum speedup you get – assuming the rest of the computation is performed in 0 time in parallel (p=1% gives max. 100x speedup).

- **Parallelism will 'only' help you solving problems with** *a fixed speedup,* **at most 10^5-10^6 times faster, but always limited by:**
  - the time you can wait for the answer
  - your parallel algorithm
  - the number of CPUs
  - the frequency of a single computing element

Conclusion: Faster calculations need:

1. **Better sequential algorithms** transformed into
2. **Better parallel algorithms**, and first then:
3. **A 'big parallel machine'**

How to parallelize a problem

- **Partition**
  a) The program
     - 'Different' parts of the algorithm on each CPU
  b) The data
     - Every CPU has its own part of the data set, but same program
  c) Partition both program and data
- **Communication**
  - Asynchronous calls
    - Send (don't wait for answer)
    - Wait for someone to call you
    - **Best**: use a separate thread to communicate at both ends
  - Synchronous calls (send and wait for answer = no parallelism)

When parallelizing – how to partition program & data

- **The program grain size:**
  - (neighbouring instructions, done by the CPU)
  - the inner part of a loop (HPF, openMP)
  - a procedure call (PRP)
  - an object (Proactive)
  - a process / subprogram (MPI, PVM)
- **The data grain size**
  - a set of XX (e.g. numbers 1 to N) is divided into n smaller sets
  - a 1D array is divided into n (equal) parts (smaller 1D arrays)
  - a 2D matrix is divided into its rows (or. columns)
  - a 2D matrix is partitioned into separate blocks
  - a 3D matrix is partitioned into separate 2D matrices

How parallel is your problem?

- **Embarrassingly :**
  – seq – parallel – seq
- **Some:**
  – seq_1 – $parallel_1$ – seq_2- $parallel_2$ – seq_3,..., - $parallel_n$ – seq_{n+1}
- **None:**
  – seq

Other issues

- **Scaling**
  - With n CPUs you want n times faster program (perfect scaling)
- **Load balancing on n CPUs**
  - If you don't divide your algorithm wisely into 'equal parts', or if some of the CPUs are slower, you might not get perfect scaling .
- **Fault tolerance**
  - One, two, many of the CPUs, or parts of the net goes down
- **Avoiding (most of) the communication delay:**
  - MultiCore CPU
  - SMPs
  - PC clusters
  - A Grid (clusters of clusters, at different locations)
- **Shared memory (R&W)**
  - Start and end of program OK
  - When running program (potentially huge bottleneck)
- **Distribution, monitoring progress**

PRP Overview

- **The idea and the project**
  – Type of recursion supported and demand on net and machines
- **Transforming a program to a PRP program**
- **What kind of problems can be solved**
  – Performance figures for 3 problems
- **Some implementation features**
  – The pre-processor and runtime system
  – Almost eliminating delay
  – Workload, unbalanced problems and fault tolerance
  – Portability
- **Conclusion**

PRP, basic idea:

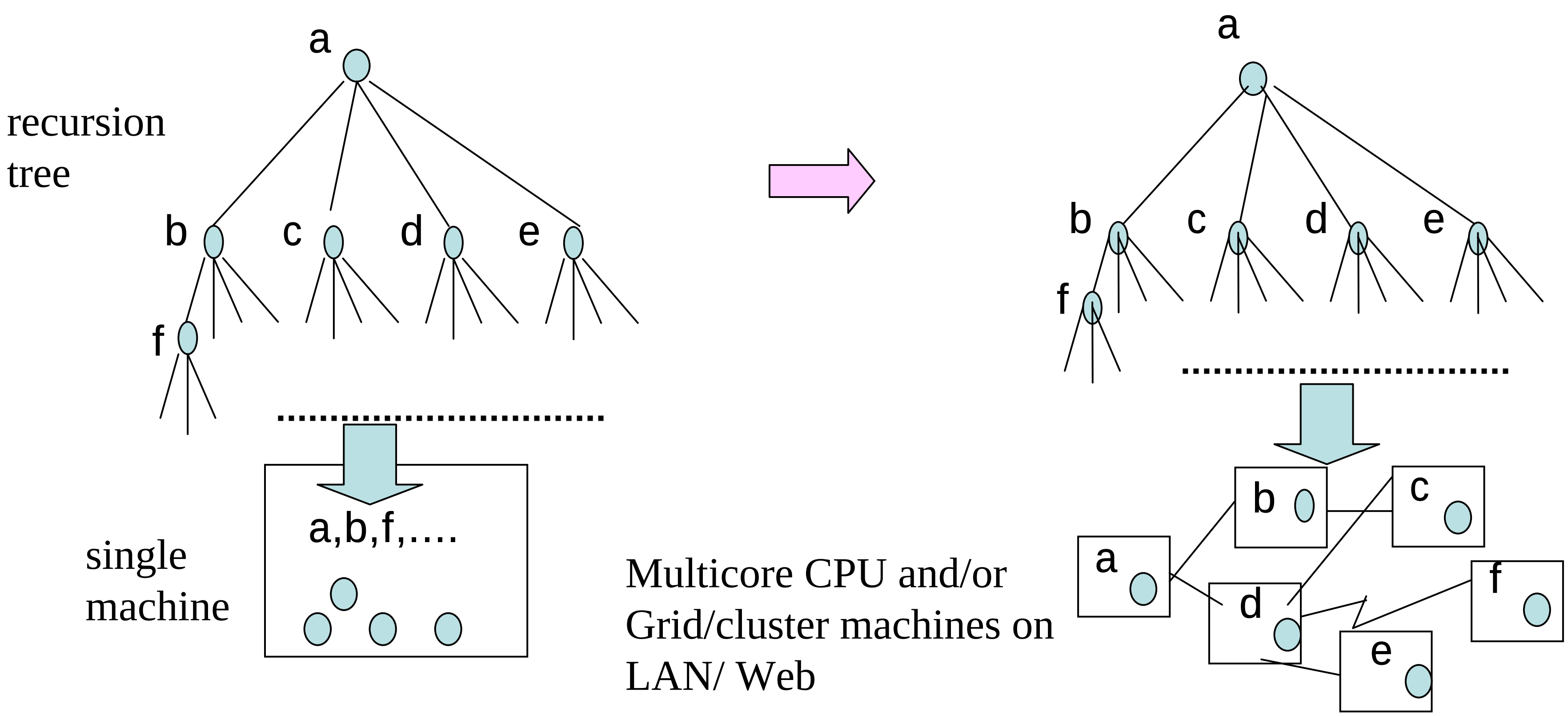
1. Take a sequential program with recursion
2. Put two comments into the program
3. Let the PRP-system compile & run it
4. Voila: The program goes n times faster on n CPUs

Does it work that way?

YES, with some limitations ...

The basic idea (cont.)

- **Take sub-trees of the recursion tree and automatically perform them on connected machines in usual fashion.**



The PRP project

- **Eleven master thesis, all made working code & improvements over previous efforts (simplifications and/or speed improvements) .**
  - Initial idea, Maus (1978)
  - Torfinn Aas, C over RPC (1994)
  - Yan Xu, C over MPI, shared memory (1997)
  - Arne Høstmark, C over PVM, fault tolerant, 100+ machines (1997)
  - Viktor Eide, C over PVM, adm/worker(1998)
  - Tore A. Rønningen, Java over RMI, buffered parameters (2003)
  - André Næss, Parallel execution of STEP/EXPRESS (2004)
  - Christian O. Søhoel, Parallel chess (2005)
  - Mats Bue, PRP on .Net in C# (2005)
  - Bjørn Arild Kristiansen, GUI and online monitoring (2006)
  - Cuong Van Truong, fault tolerant, unbalanced recursion trees (2007)
  - Jørn Christian Syvertsrud, Load balancing, repeated recursion (2007)
  - [Kristoffer Skaret, Multicore CPU (2008)]
  - [Daniel Treidene, , Loop-parallelization (2008)]

Demand on recursive PRP-procedure/method

1. **The recursive call is textually only one place (in some loop)**
2. **No return value from a method can be used to determine parameters to another call**
3. **All information used by a PRP-method must be in parameters (no global variables – except for multicore), but constants can be declared and used in local class and local data can be declared and used.**
4. **Code in the procedure from start to recursive call must be repeatable (same result second time performed).**
5. **Any parameter and return type possible from a the PRP-method - also objects. But if these classes are declared in your program, they must be stated as: '**implements Serializable**'.**
6. **Obvious: The fan-out (number of recursive calls) per method must be (on the average) >1**

JavaPRP - The pre-processor and runtime system

- **The original program is tagged with 2 or 3 comment-tags:**

  /*PRP_PROC*/ - meaning: next line is declaration of the recursive procedure

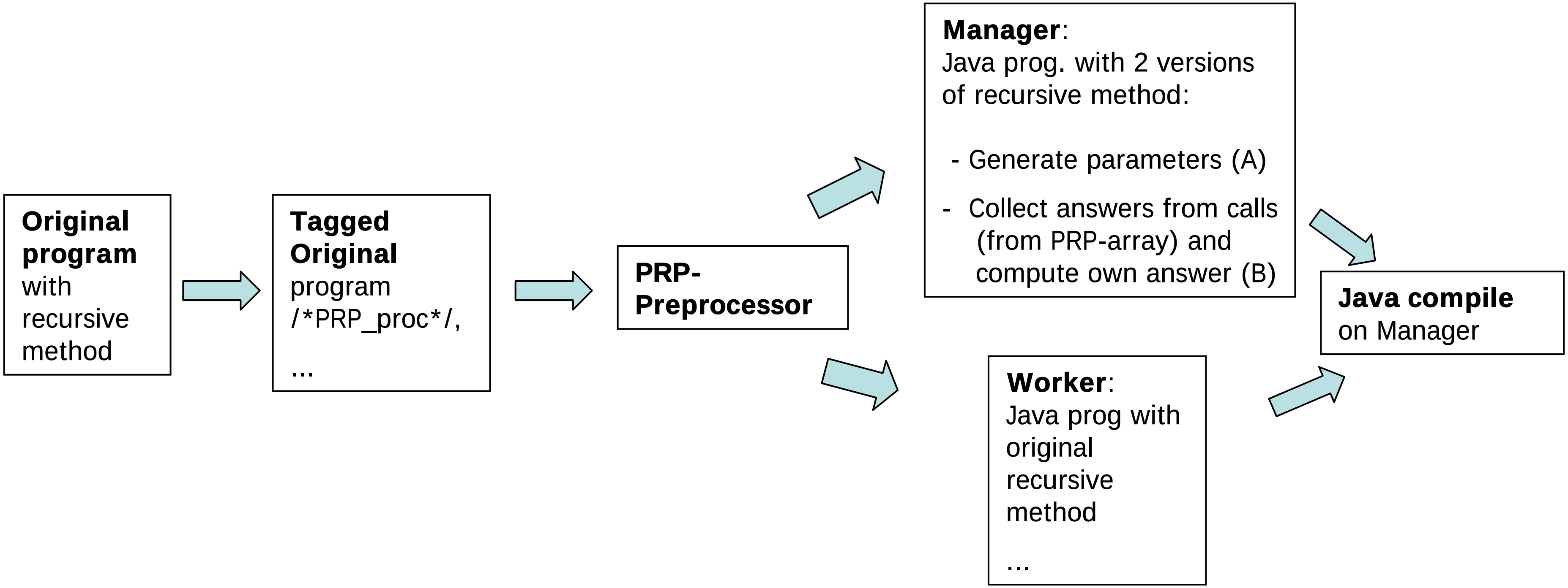
  /*PRP_CALL*/- meaning: next line is the recursive call (in a loop).

  /*PRP_FF*/ - if on first line in program, signals that the recursive proc. only has one recursive level (full fan-out). Optimizing option only.

- **The pre-processor splits the Program into two versions:**
  - **Manager**: The master node, starts program, splits the recursion tree, generates parameter sets, receives return values.
    - Has two versions of the recursive procedure, one for 'breaking up' the recursive tree , one for collecting results and finish the calculations.
    - Handles also: Initial distribution of code to workers, load balancing, fault recovery, GUI monitoring, termination.
  - **Worker** code: Receives in succession a number of parameter sets from Manager, computes (ordinary recursion, top-down), returns answer.

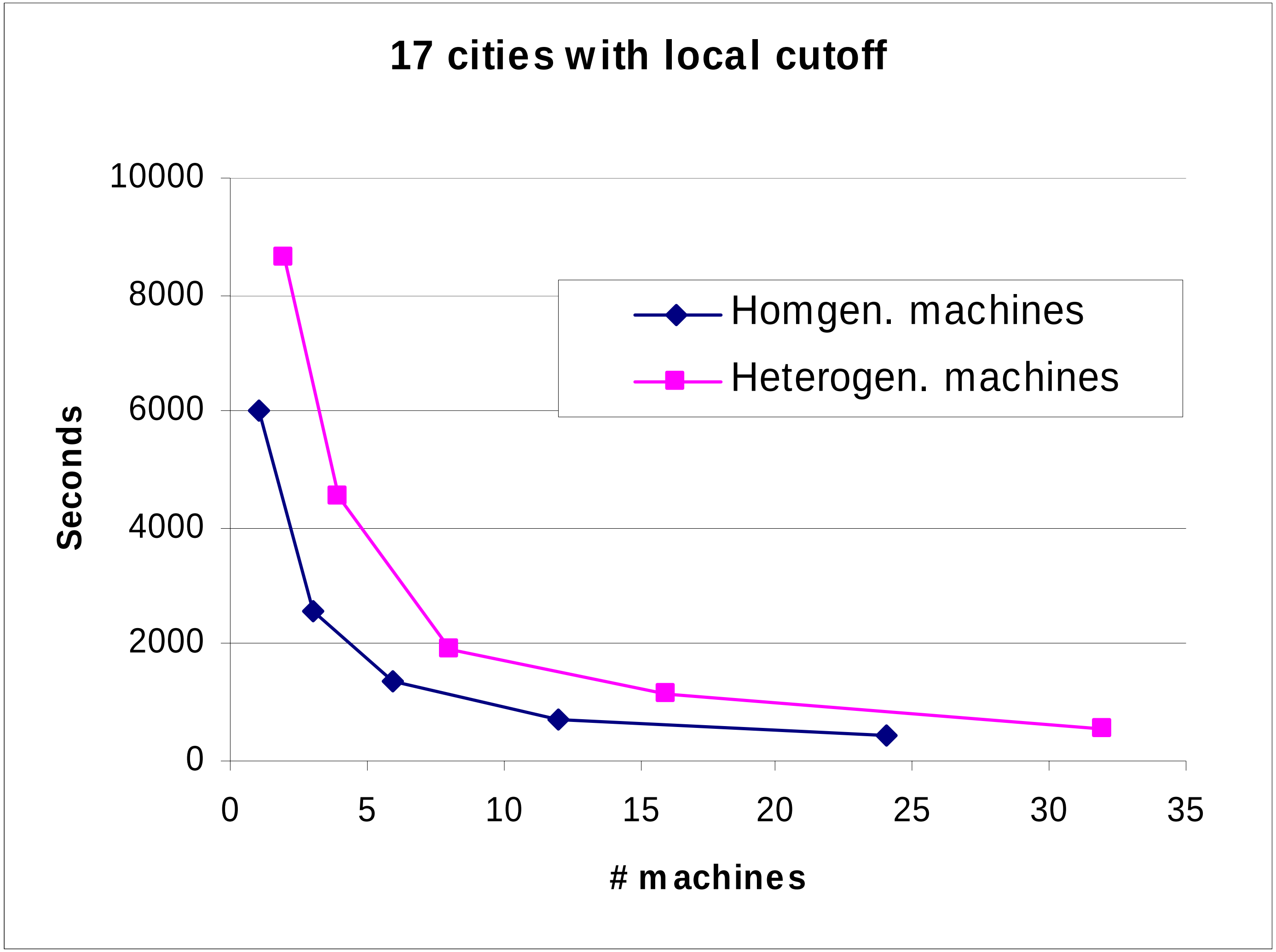
How to compile and run PRP

- You are logged on the Manager machine and on (say by ‘telnet’) on M worker machines and have started the worker-runtime system on each worker.
- Then start Manager program, who will spread code to Workers, start recursion in main, send sub problems to Works, collect answers, print result,.. as specified in the original program.



```java
public class tsCutoff {

  public static int byer[][] =
  {{0,633,257,91,412,150,80,134,259,505,353,324,70,211,268,246,121},
   {..............................................
}};
   public static boolean[] brukteByer = new boolean[17];
   public static int kortest = Integer.MAX_VALUE;

   /*PRP_PROC*/
   public static int tsCut(int denne,  boolean[] brukt, int lengde){
      int svar;
      boolean sisteBy = true;
      if(lengde>=kortest) return Integer.MAX_VALUE;

      for(int i=0;i<17;i++){
         if(!brukt[i]){
            brukt[i]=true; sisteBy = false;
            /*PRP_CALL*/
            svar = tsCut(i,brukt,lengde+byer[denne][i]);
            brukt[i]=false;
            if(svar<kortest) kortest=svar;
         }
      }
      if(sisteBy) return lengde+byer[denne][0];
      else return kortest;
   }

   public static void main(String[] args) {
      long startTid;
      startTid=System.currentTimeMillis();
      brukteByer[0]=true;
      System.out.println(tsCut(0,brukteByer,0));
      System.out.println(System.currentTimeMillis()-startTid);
   }
}
```

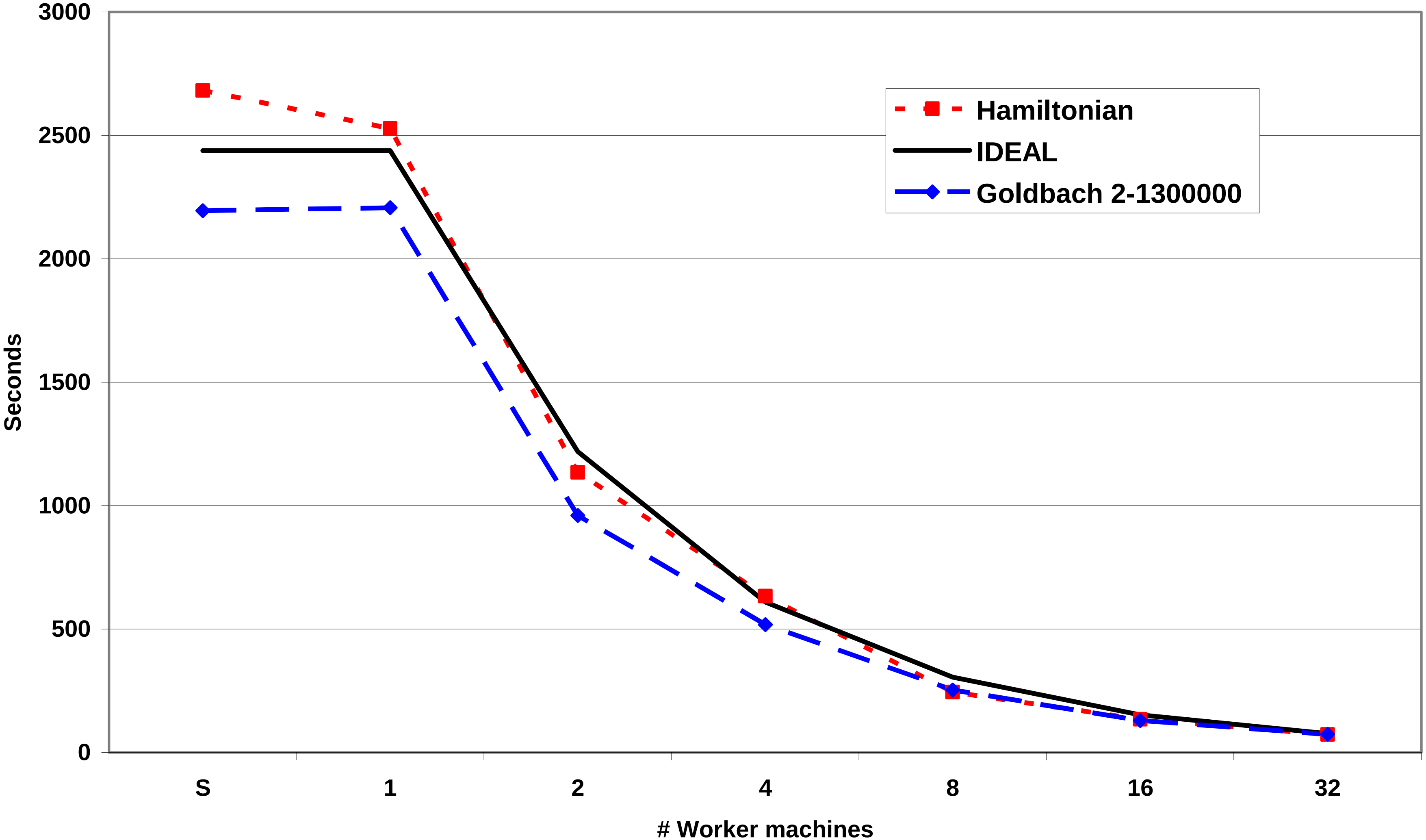
Travelling Salesman – 17 cities with local cutoff value in each Worker machine



I) Two other simple problems - scaling test (# machines = 2 to 32)

- **Number of distinct Hamiltonian Circuits in a Graph:**
  - 20 nodes, each with from 3 to 9 edges.
  - Recursive decent,
- **Goldbach's hypothesis:**
  - How many distinct pair of primes , p1 + p2 = n, are there for all even numbers n = 2, 4,6,..., 1300 000
  - Takes almost as long time as the Ham. Circuit instance on one machine
  - Data partition – full fan-out from first method called
- **Compare to Ideal curve (perfect scaling):**
  - TimeOnOneMachine/(# machines -1)

Data partition/full fanout (Goldbach) and recursive decent (Hamiltonian) problems compared with IDEAL: T1 / (#machines - 1)



PRP Implementation – overview:

1. **Generate enough parameter sets in one machine (the Manager)**
2. **Parallelize**
   - Send out these parameter sets to different CPUs (Workers)
   - Get answers back and put results in a result array until all parameter sets are solved
3. **Perform in Manager those calls that generated the parameters – the top of the rec. tree**
   - and return the result from the top call to the original call (in `main`)

1. Implementation: catch parameter sets:

0. **Make a variant A of the recursive call where the recursive call are substituted with a method call to: `stealParameters` – with the same parameters (and the rest of the code is removed).**

1. **The parameter set to the first call (from `main` to the** recursive procedure**) is put in a FIFO-queue.**

while (<not enough parameter sets in the FIFO>) {

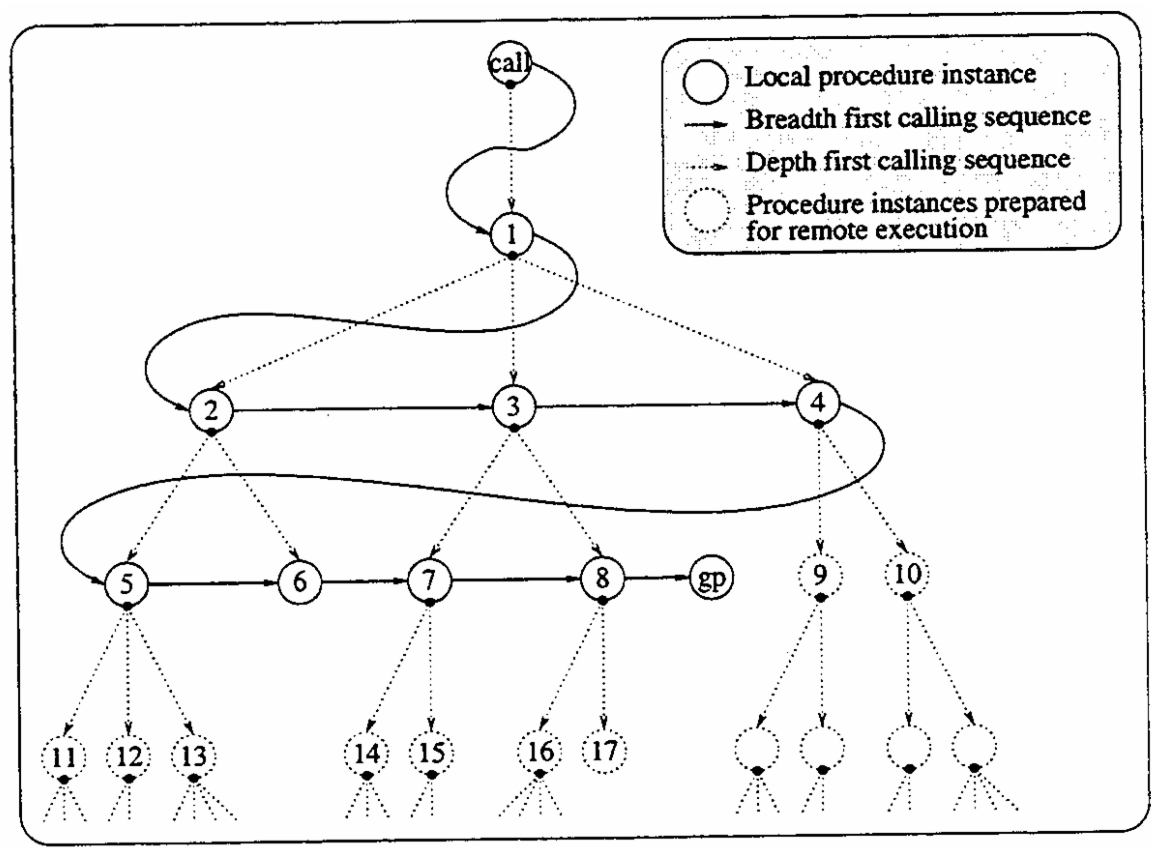
2. **Remove first element in FIFO, and put this parameter set on a software call-stack. Then call A with this parameter set.**
3. **This generates more calls to `stealParameters` and each call will put its parameter set in the FIFO(=tasks for parallelization)**

}

Now those calls that generated parameters for parallelization are on the call-stack (the top of the tree).
The parameters for parallelization are in the FIFO

Manager: To deal with different machines (fast & slow), varying workload and unbalanced problems:

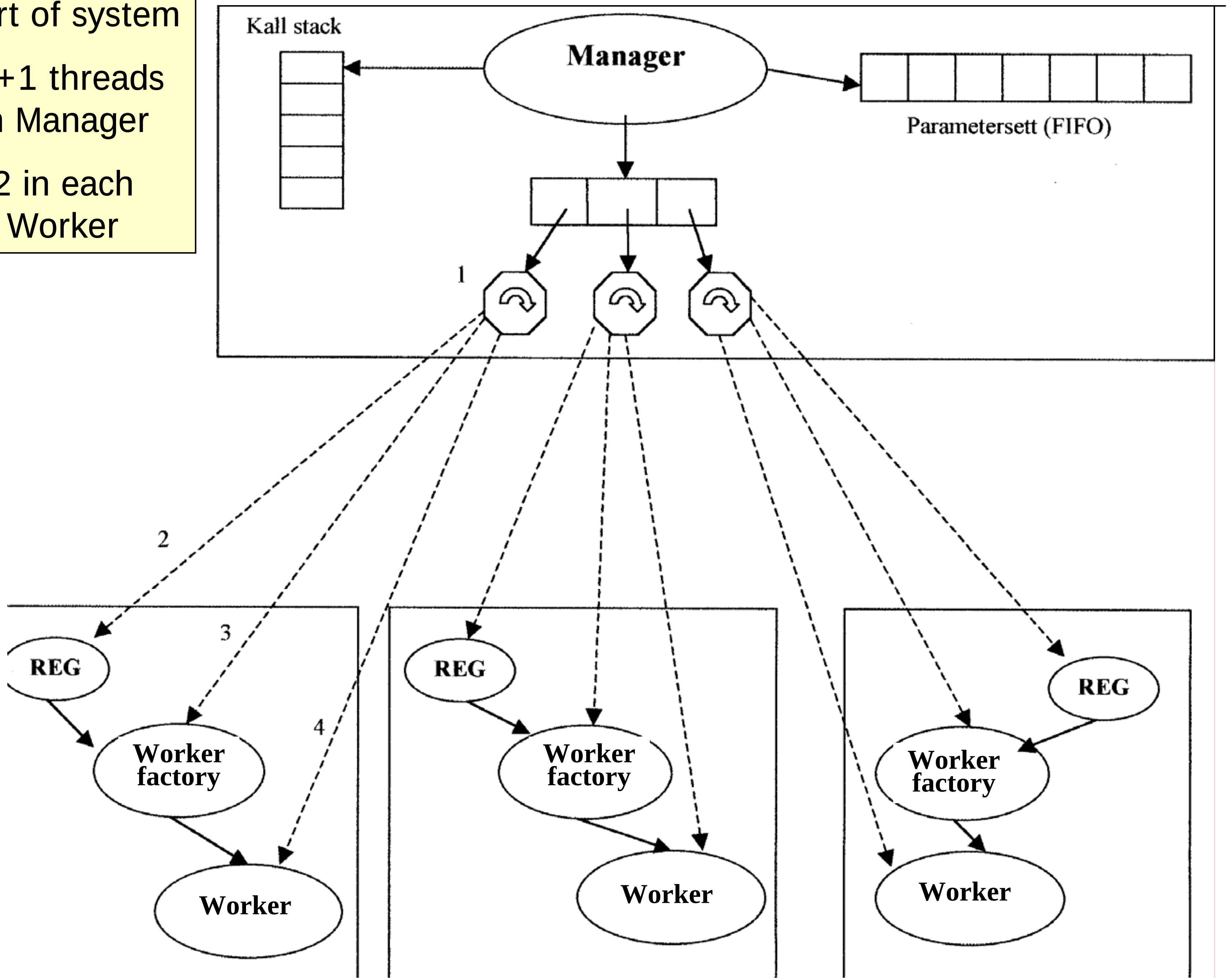
- Generate lots of parameter sets to workers (20*numWorkerMachines)
- Run procedures up to recursive call – put parameter sets on FIFO
- One thread per worker (send parameters <-> receive result)



Start of system

n +1 threads in Manager

2 in each Worker

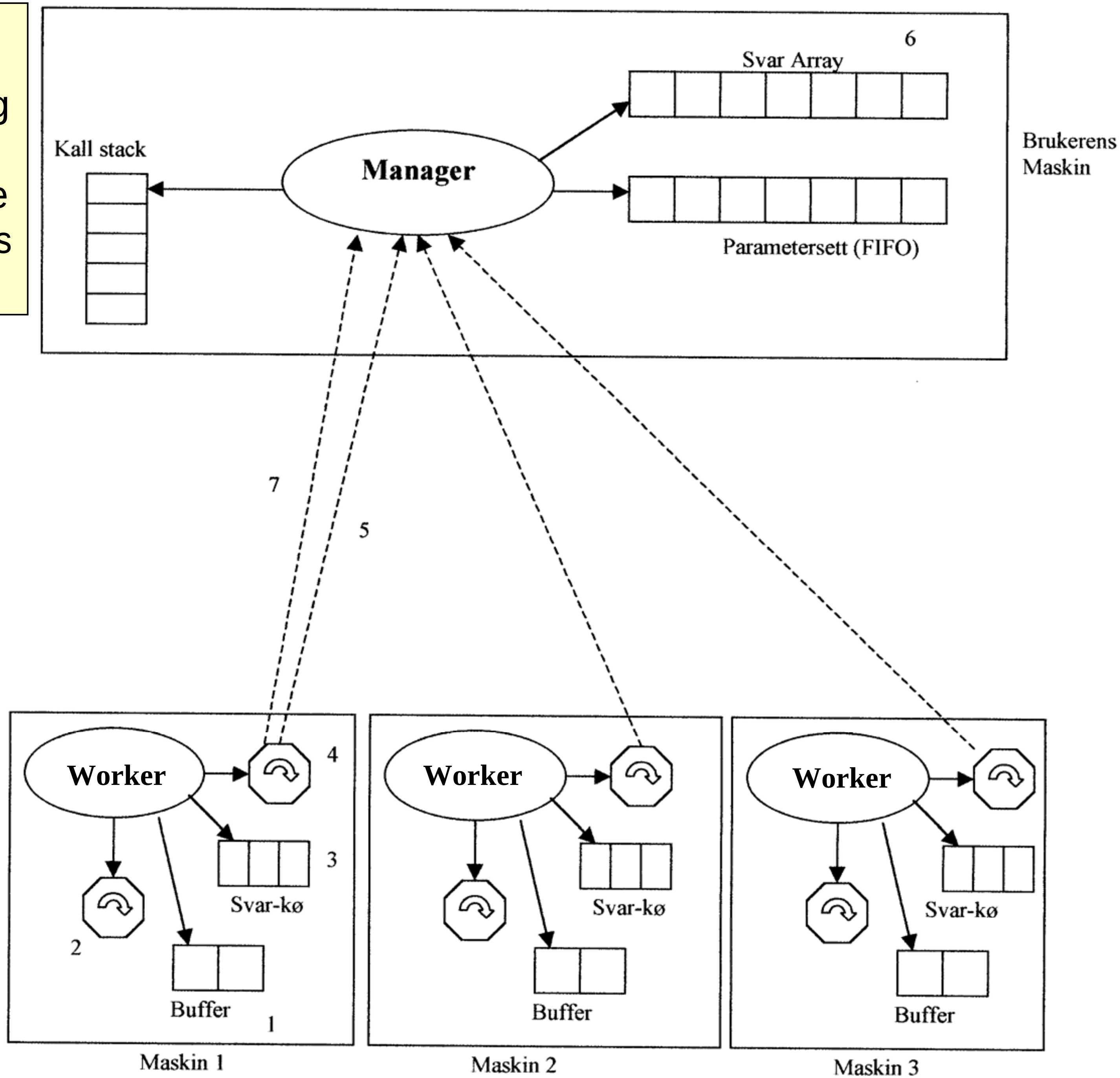


Why do more parameter sets help?

- **Since, on the average there are 20 sub-problems per Worker-machine, we get workload balancing:**
  - Those who finishes one parameter set, starts a new set from the buffer immediately,
  - and get a next parameter in the buffer as soon as return values are received by Manager for previous sub problem
  - If '**`data transfer time' < 'CPU-time`**' for the sub-problems, the net does not matter for the total compute time because of buffering.
- **When there are no more *new* problems to fan out:**
  - Those calls who hasn't answered yet, are re-sent to fastest, idle machine so far (solves error on net & machines problems)
  - The first returned answer to a problem is taken – other calls with the same parameter set are then 'killed'.

2. System running

Workers performing the ordinary recursive procedure with the parameters sent over the net



Avoiding delay from the net – buffering of 2 tasks.

A) Usual send/receive (Worker waits two net delays: 3-5)

Manager ← - - - - - - → Worker

1) M: Send $task_1$
2) W: receive & do $task_1$
3) W: Send $result_1$
4) M: Receive $result_1$
5) M: Send $task_2$
6) W: receive & do $task_2$

....

B) Buffered send/receive (send two tasks first time)

after start up, no wait on net if compute time > transfer time

Manager ← - - - → Worker

1) M: Send $task_1$ & $task_2$
2) W: $receive_{1\&2}$ & do $task_1$
3) W: Send $result_1$ & do $task_2$
4) M: Receive $result_1$ & send $task_3$
5) W: Send $result_2$ & do $task_3$

....

3. Last step of a PRP computation:
Perform the top of the recursion tree (now on the stack):

A second variation of the recursion procedure, B, is made in the Manager where the recursive call is substituted with a call to `pickUpAnswerInResultArray()`

```
while(<more calls on software stack>) {
```

1. Remove top of software stack.
2. Call B with its parameters (this is the second time this recursive procedure is called).
3. This generates calls to **`pickUpAnswerInResultArray`** were its answers are found.
4. The rest of B is performed and its answer is put into the Result Array.

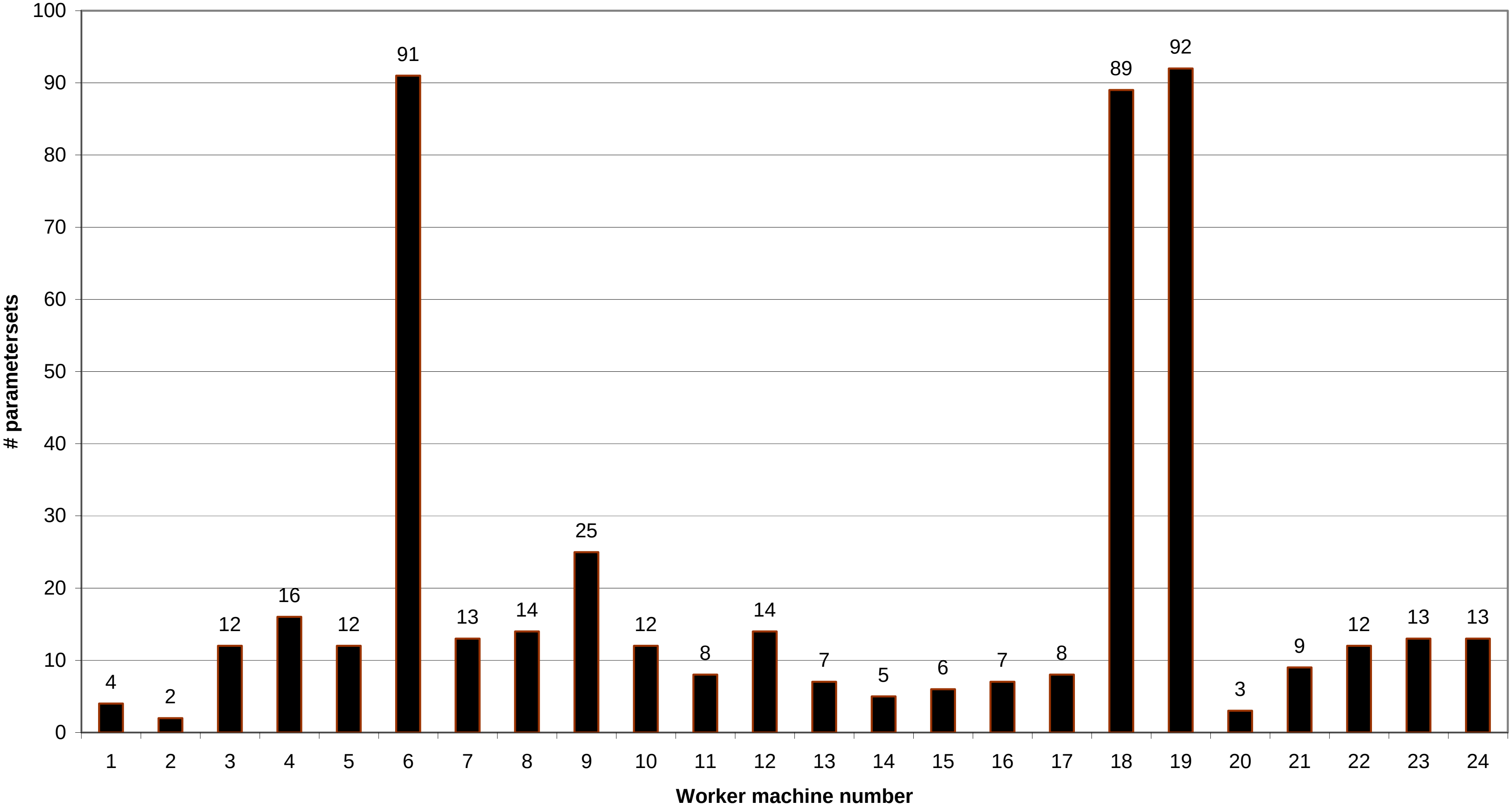
```
}
```

The result from the first call (the bottom last on the call-stack & in pos 1 in Result Array) is returned to the call from **`main`**

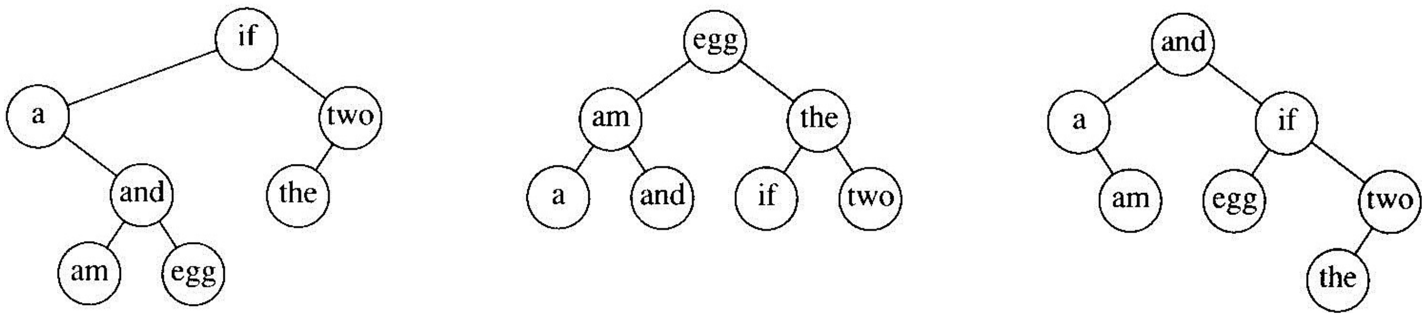
What kind of problems can be solved (n = amount of data)

- **The problem has to be execution time $O(n^{1.5})$ or harder (and running time > 1 minute)**
  – No point in parallelizing easier problems !
- **NP-problems in graphs or trees (branch and bound):**
  – Example Travelling Salesman, Hamiltonian Circuit, ...
  – With or without 'global' variables for cut off.
- **Data division problems in general:**
  1. If we have to run through data: d1,d2,....dn, then recursively divide the dataset into 20*(number of machines) equal parts and call each in a loop (=full fan-out recursion).
  2. Solve problem on each part in a PRPproc instance – then return and combine
  3. Example : Optimal binary search tree – $O(n^{1.5})$ (M.A.Weiss, p. 379)
- **Other CPU intensive problems:**
  – Prime number counting, Goldbach's hypothesis (**2i = p1+p2**, i =2,3,4,..)

Number of parameter sets handled by each Worker machine - 24 equal machines



III) The optimal binary search tree – (from V.Eide's thesis & Kristoffer Skaret)

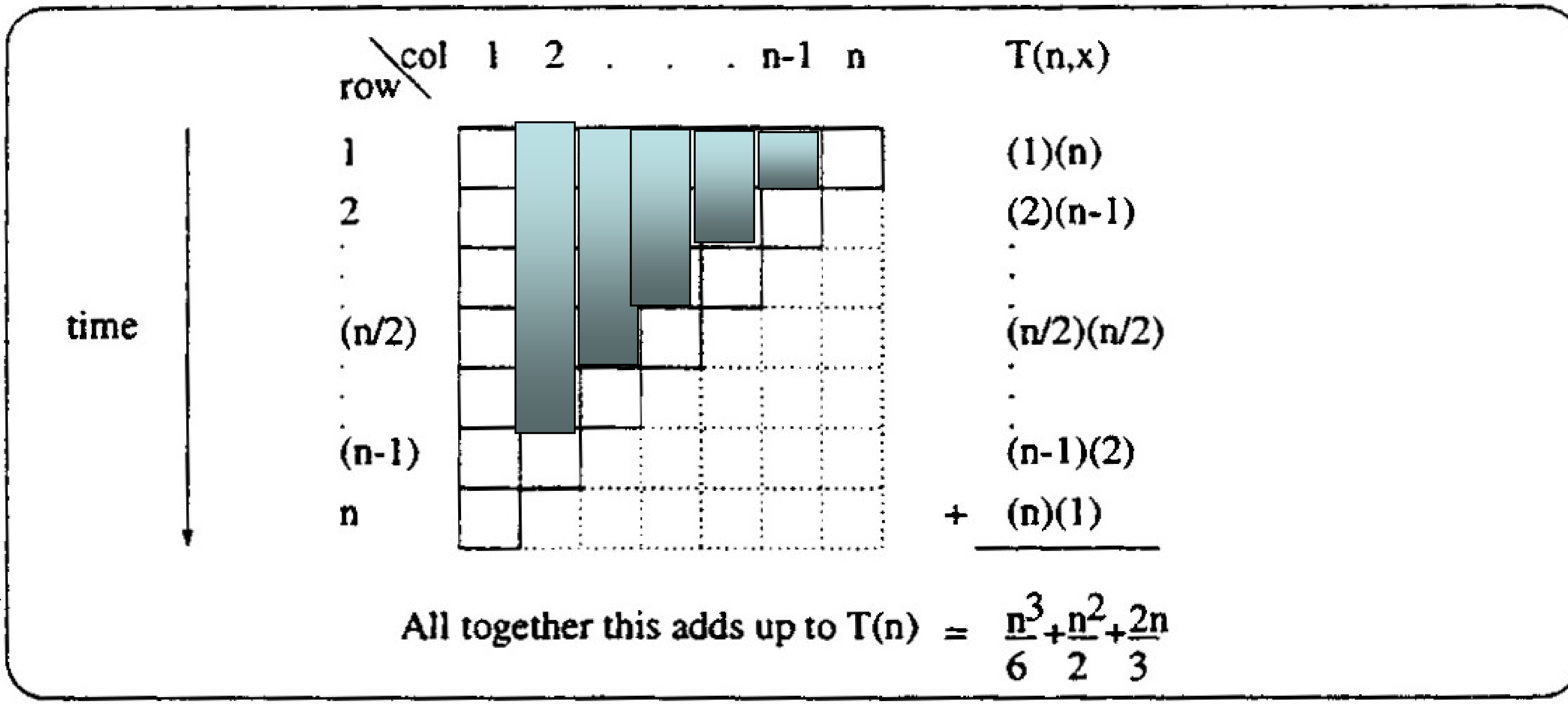


Input		Tree #1		Tree #2		Tree #3	
Word	Probability	Access Cost		Access Cost		Access Cost	
w_i	p_i	Once	Sequence	Once	Sequence	Once	Sequence
a	0.22	2	0.44	3	0.66	2	0.44
am	0.18	4	0.72	2	0.36	3	0.54
and	0.20	3	0.60	3	0.60	1	0.20
egg	0.05	4	0.20	1	0.05	3	0.15
if	0.25	1	0.25	3	0.75	2	0.50
the	0.02	3	0.06	2	0.04	4	0.08
two	0.08	2	0.16	3	0.24	3	0.24
Totals	1.00		2.43		2.70		2.15

Data structure – compute optimal binary search trees for all subsections of data – finally for all data

	Left=1	Left=2	Left=3	Left=4	Left=5	Left=6	Left=7
Iteration=1	a..a .22 a	am..am .18 am	and..and .20 and	egg..egg .05 egg	if..if .25 if	the..the .02 the	two..two .08 two
Iteration=2	a..am .58 a	am..and .56 and	and..egg .30 and	egg..if .35 if	if..the .29 if	the..two .12 two	
Iteration=3	a..and 1.02 am	am..egg .66 and	and..if .80 if	egg..the .39 if	if..two .47 if		
Iteration=4	a..egg 1.17 am	am..if 1.21 and	and..the .84 if	egg..two .57 if			
Iteration=5	a..if 1.83 and	am..the 1.27 and	and..two 1.02 if				
Iteration=6	a..the 1.89 and	am..two 1.53 and					
Iteration=7	a..two 2.15 and						

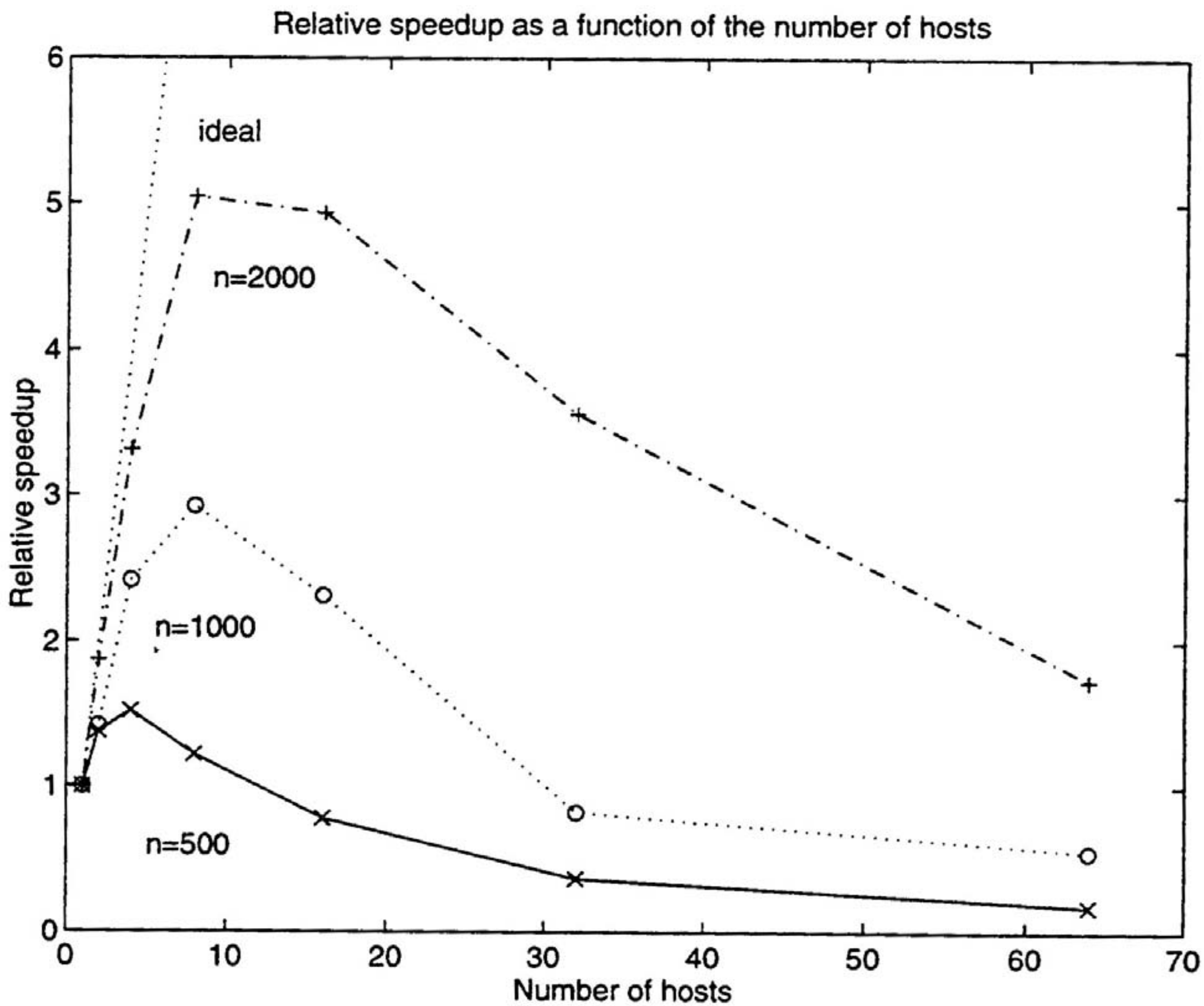
T(n,x) = work for computing next line,
= what you need to know to compute ‘p’



This problem is $O(n^{1.5})$ since we have $n*n= n^2$ data and execution time n^3

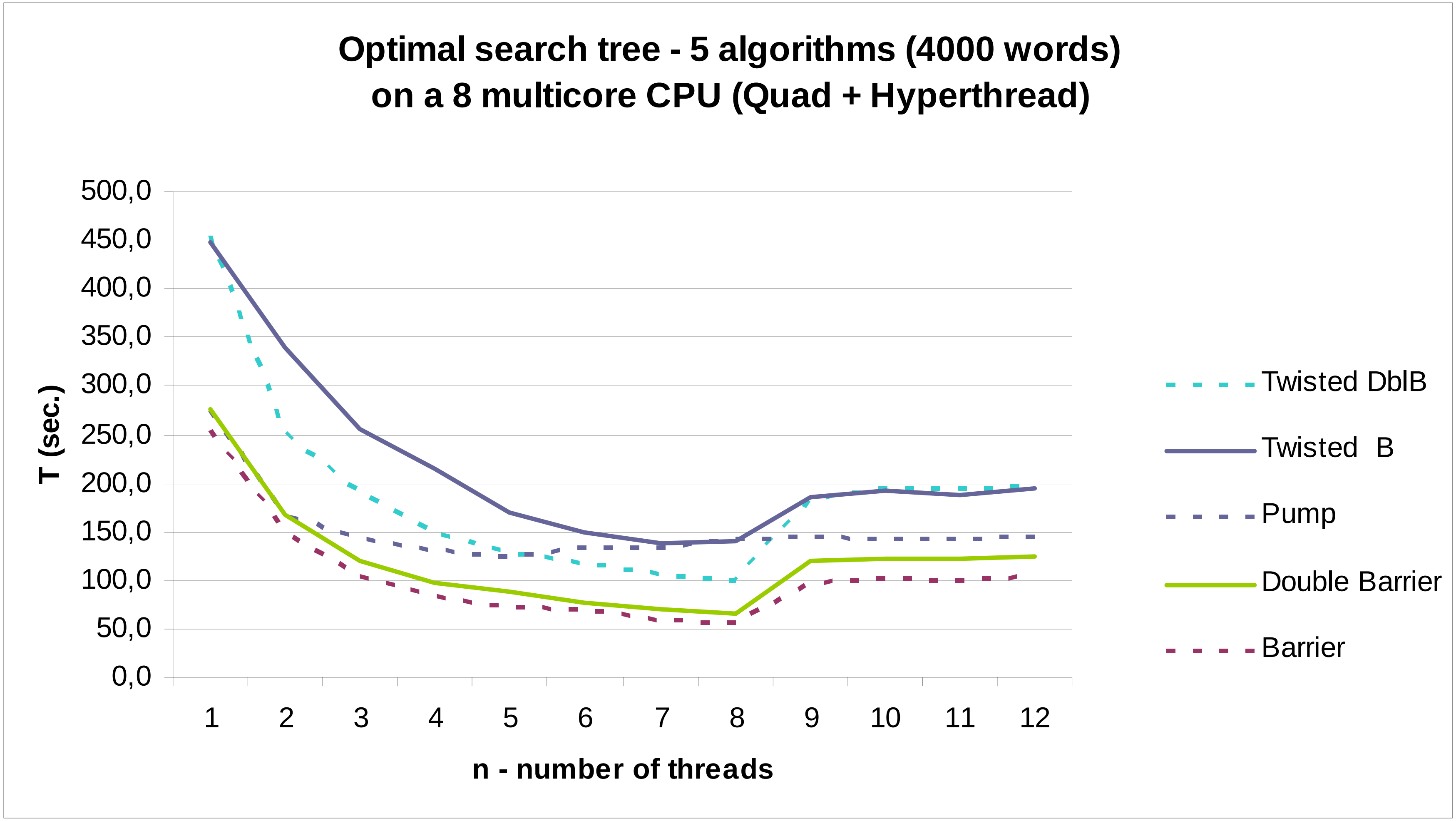
Words in instance, n.	sequential	parallel	number of hosts					
		1	2	4	8	16	32	64
500	6.7	11.7	8.5	7.7	9.6	15.0	31.8	65.2
1000	106.6	117.6	82.9	48.6	40.2	50.4	143.0	213.2
2000	1068.8	1265.1	674.7	381.6	250.7	256.3	355.2	731.6

Results and speedup



Current research: Finding PRP implementation for Multicore CPUs

Different matrix organization (twisted = row /no = diagonal)
Different synchronization (double = read, sync. write, sync/not= read&write, sync)
Pump (Central start thread, read write, return)



Conclusions

- **PRP: A package for using recursive methods in Java (and C) as a tool for parallelizing programs.**
- **More high level than most parallelizing libraries**
- **Little modification to original code is needed:**
  - Put two comments into code, run pre-processor and compile
  - Some restrictions on the recurcive procedure.
- **Scales (almost) ideally for NP-type problems where ratio: Data-sent/compute-time is very small**
- **Handles also (one) problem of $O(n^{1.5})$, where data sent over net in sum is O(n) reasonably well - scales up to factor 4-5**
- **PRP also used on a real industrial problem with good speedup**
- **Now, a special implementation for multicore CPU and loop parallelization**

PRP is a general tool for parallelizing a number of recursive algorithms – or many calls to the same procedure in a loop.