Chapter 4

Case Studies

Before we will examine Pangaea's behavior in several actual application programs, it is useful to study some properties of the *platforms* that Pangaea uses individually. One such study was performed by Miriam Busch (2001) to assess the performance of local invocations, remote invocations, and object migration on the RMI-based JavaParty and Doorastha platforms, and RMI itself. To this end, dummy objects were invoked in parameterless calls without return values, both across the network and locally. In particular, the times were measured for:

a remote call on:

- a *global* (remotely invokable) object,
- a remote, *migratable* object that has not yet migrated,
- a *migrated* object, which started out locally and was then migrated away to another machine, and

a local call on:

- a *local* object (without any distribution technology intervening),
- a *global* (remotely invokable) object on the same machine,
- a *migratable* object that still resides on the local machine, and
- a *returned* object that was migrated *back* to the local machine.

For RMI itself, migration is not available and so some cases do not apply. The last case, a *returned* object was "simulated" by looking a de-facto local object up in the name server, and then invoking it.

Measurements were performed using Sun JDK 1.3 under Linux 2.4, JavaParty 1.04e and Doorastha version 2.2.3 from the respective projects' home pages. For JavaParty, both the RMI-based "reference implementation" and the optimized version based on KaRMI (Philippsen et al. 2000) was used. The hardware consisted of two equivalent Pentium III Coppermine 700 MHz PCs with 256 MByte RAM, connected via a closed loop, twisted pair Ethernet at 10 and 100 MBit/s. Each call was performed 10,000 times, and the execution time then divided by the same number to obtain the results shown in table 4.1.

The execution times range from 700 μs for the most expensive remote call to 50 ns for a local call, which is a factor of 14,000. The table also shows that potential migratability has no significant effect on the performance. What is noteworthy is however the relative performance of remote and de-facto local calls, which is shown graphically in figure 4.1.

A remote call with RMI as the underlying mechanism takes about 700 μs on 10 MBit/s on all platforms, which is reduced to about 260 μs on 100 MBit/s. The optimized KaRMI layer

Call Type	Receiver	Network (MBit/s)	RMI	JP (RMI)	JP (KaRMI)	Doorastha
remote call	global object	10	679.2	700.5	462.9	686.8
		100	259.5	262.1	116.1	267.6
	migratable object	10	—	676.9	465.7	681.6
		100	—	265.1	116.0	267.2
	migrated object	10	—	675.6	461.4	685.8
		100	—	264.0	116.8	267.2
local call	local object	—	0.05	0.05	0.05	0.05
	global object	—	0.05	256.7	51.1	0.05
	migratable object	—	—	257.6	51.0	0.07
	returned object	—	208.9	259.7	51.6	0.09

Table 4.1: Times in μs for a single call

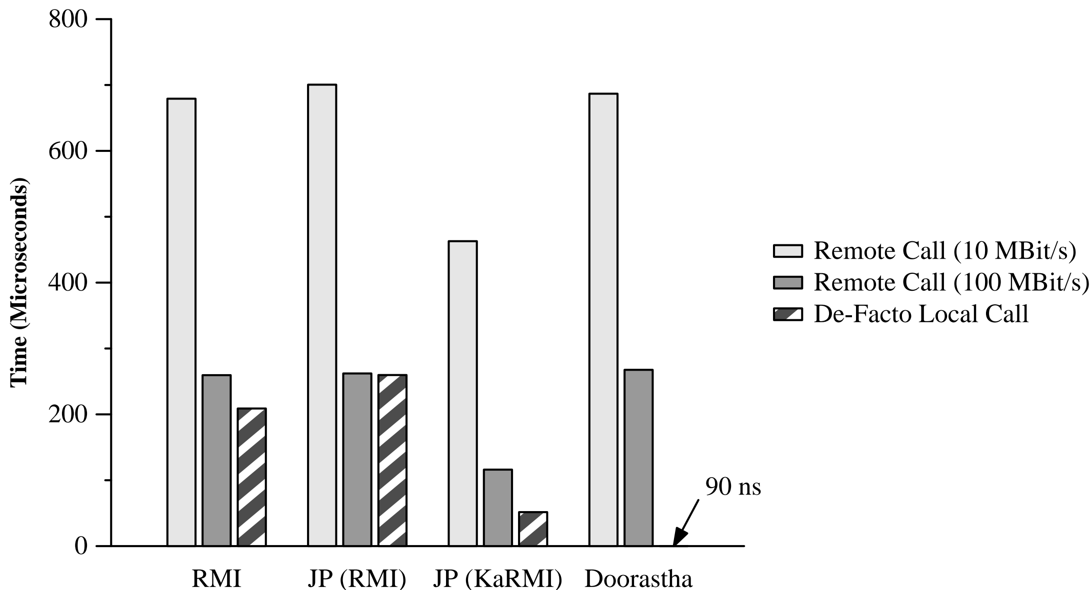

Figure 4.1: Platform Performance

achieves 460 μs and 116 μs, respectively, corresponding to 66% and 57% of RMI's execution time. However, a de-facto local call on a returned object takes essentially the same time as a remote call on a fast network with RMI, and still 51μs on KaRMI, which is about 20% of the remote case on a fast network. This is still about 1,000 times slower than a local invocation. With Doorastha, on the other hand, this call takes a mere 90 ns, which is less than twice the time for an ordinary local call.

Despite its significant optimization, KaRMI obviously does not bypass the proxy completely for a returned object, while Doorastha does. The difference amounts to three orders of magnitude, which renders object migration essentially useless with KaRMI. To be fair, however, it must be said that JavaParty/KaRMI maintains remote invocation semantics after migration, which Doorastha cannot do in the general case. This points to a much more severe language design issue which we will return to in chapter 5. In our case studies involving the migration subsystem, we will however restrict ourselves to the Doorastha platform.

In the following, we will demonstrate the Pangaea system in three case studies. For each of the example programs, we will use the following structure to describe them and their distribution:

Program Analysis The object graph created by Pangaea's analyzer, and its properties. This section makes it clear how the program actually works, and how well this is captured in the object graph.

Distribution Analysis This section argues what would be good, i.e. efficient distributions for the program, and how close Pangaea's automatic distribution algorithms, and the back-end systems used, come to finding and realizing these distributions.

Performance The performance of the distributions found in the previous section is shown and analyzed.

4.1 RC5

The RC5 program is a classical manager/worker application that breaks RSA encryption keys by a brute force search of the key space[1]. The program consists of three classes and an interface, totalling about 350 lines of code.

Program Analysis

Pangaea computes the object graph of RC5 in 4.4 seconds (see page 72 for more details); the result is shown in figure 4.2.

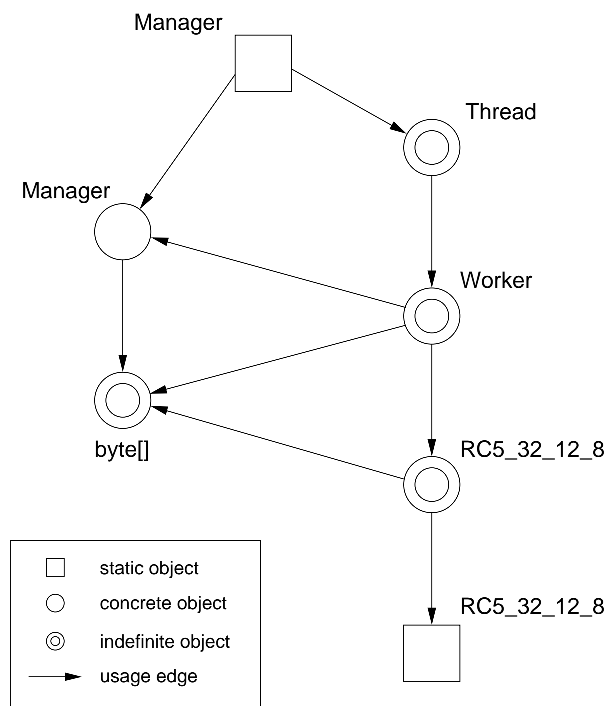

Figure 4.2: Object Graph of the RC5 Program

The program is started by the main method in the static part of the *Manager* class. This method creates a dynamic *Manager* object, the *Worker*s, and *Thread*s to run them. The unit of work is a byte array of length 8, which represents the first key that a *Worker* should try. These byte arrays are created by the *Manager*, and retrieved by the *Worker*s as the result of a

[1]The RSA implementation was written by Greg Hewgill; we added the manager/worker logic ourselves.

method call. Upon receiving the byte array, the *Worker* tries a fixed number of keys starting with the one represented by the byte array, and reports any success back to the manager. The actual RC5 algorithm is encapsulated in an instance of the class *RC5_32_12_8*, which in turn relies on a number of constants defined in the static part of this class.

For clarity, we show the main loop of the *Worker*s' computation below. One potential difficulty for the analysis is successfully handled here by Pangaea: The RC5 algorithm is not called directly, but via the *Algorithm* interface. However, since only the *RC5_32_12_8* object is available to the *Worker* as an implementer of that interface, the Analyzer correctly infers the usage relation shown in the object graph.

```
public void run() {
  Algorithm algorithm = new RC5_32_12_8();
  byte[]     key;
  while (true) {
    key = manager.getWork();
    if (key == null) {
      // if there's no work for us, we're done
      break;
    } else {
      for (int i=0; i < iterations ; i++)
        if (algorithm.isCorrectKey (key , ...)) {
          manager.reportSuccess (key);
        } else {
          // increment key
          for (int j=0; j < keySize; j++) {
            key[j] = (byte)(key[j]+1);
            if (key[j ] != 0) break;
          }
        }
    }
  }
}
```

Distribution Analysis

As for any master/worker application, it is best to distribute the workers evenly across the processors, so that they handle the workload in parallel. Thus, each *Thread* and corresponding *Worker* and *Algorithm* object should be placed onto a new processor. The constants in the static part of *RC5_32_12_8* can be copied ("replicated") for each *Worker* since they are immutable. It would in fact be a prohibitive bottleneck for the program if a single, remotely accessible static object was used. However, this significant optimization could not have been found by any automated system except by static analysis.

Another issue is how to handle the byte arrays that represent chunks of work. These arrays are created and initialized by the *Manager*, and then used by one of the *Worker*s. The code of the method *Manager.getWork()*, which is responsible for this, looks as follows:

```
public synchronized byte[] getWork() {
  byte[] work = new byte[8];
  for (int i=0; i < 8; i++) work[i] = key[i];
  increment (key);
  return work;
}
```

To a programmer, it is clear that the returned byte array is a local object in this method: after *getWork()* returns, the array is no longer accessible to any object other than the *Worker*

that receives it. We may therefore use pass-by-value to return the array to the *Worker*, which results in the most efficient implementation.

Pangaea's static analyzer does however not use data flow analysis, so it is not able to detect the above optimization. It would in fact be an excellent case for factory method detection as in cJVM (see page 47), which would be a worthwhile addition to our Analyzer. Pangaea in its current form can only use a second-best guess, turning the array into a migratable object and letting the run-time system move it to the *Worker* as soon as the *Worker* starts accessing it frequently. One might also argue that pass-by-move is a good default passing mechanism for arrays, since they tend to be accessed frequently after being received. In this case, the array would be moved to the *Worker* immediately.

If object migration is not available, and neither a programmer nor a static analyzer detects that pass-by-value can be used, the only solution is to allocate the arrays statically on the *Manager* partition, and let the *Worker*s access them remotely. This, of course, results in the worst performance.

Pangaea's automatic graph assignment scheme is sufficient to assign the *Worker*s and their corresponding *Thread* and *Algorithm* objects to an abstract partition, to keep the *Manager* on the root partition, and to replicate the *RC5_32_12_8* constants on each partition. By default, the byte array object will be assigned to the Manager partition as well; this can be varied by the programmer to achieve most of the program variants we have indicated above.

Performance

We executed the RC5 program on the Distributed ASCI Supercomputer DAS-2 in Amsterdam. It consists of 72 Linux PCs equipped with two 1 GHz Pentium III CPUs and 512 MByte RAM each. The nodes share a common file system and are connected by Fast Ethernet (100 MBit/s). For our measurements, we used IBM JDK 1.3.0 and Doorastha version 2.0.

Since the DAS-2 nodes feature dual-processor boards, we had the opportunity to compare the distributed programs to shared-memory parallel versions with two independent CPUs as well. The measurements show that the IBM JDK we used does indeed schedule threads to both processors. For the distributed programs, we always scheduled a single application thread to each node, although the dual-processor boards are likely to have an impact on the performance of the RMI communication layer.

For the measurements, the program's workers operated on chunks of 65,536 keys, and we adjusted the starting point of the search so that the correct result was found after 539 chunks. As can be seen from the source code, the manager accesses each key array exactly eight times to fill it with values. The worker extracts these values in eight individual accesses for each key (the actual RC5 algorithm does not use the array), but it also increments the key array in situ, resulting in about $65,536 \cdot 9$ array accesses for each chunk.

Altogether, we tested four different versions of the program:

- centralized (unmodified source code) with all worker threads being scheduled to the two on-board processors of a node,
- distributed, passing the arrays to the workers by value,
- distributed, passing the arrays wrapped into an object by-copy (we would have liked to test by-move but couldn't due to a problem in Doorastha; we don't expect any significant performance difference though),
- distributed, letting the arrays be moved to the workers by the asynchronous migration subsystem. For this version, we varied the number of calls after which a migration decision

was taken.

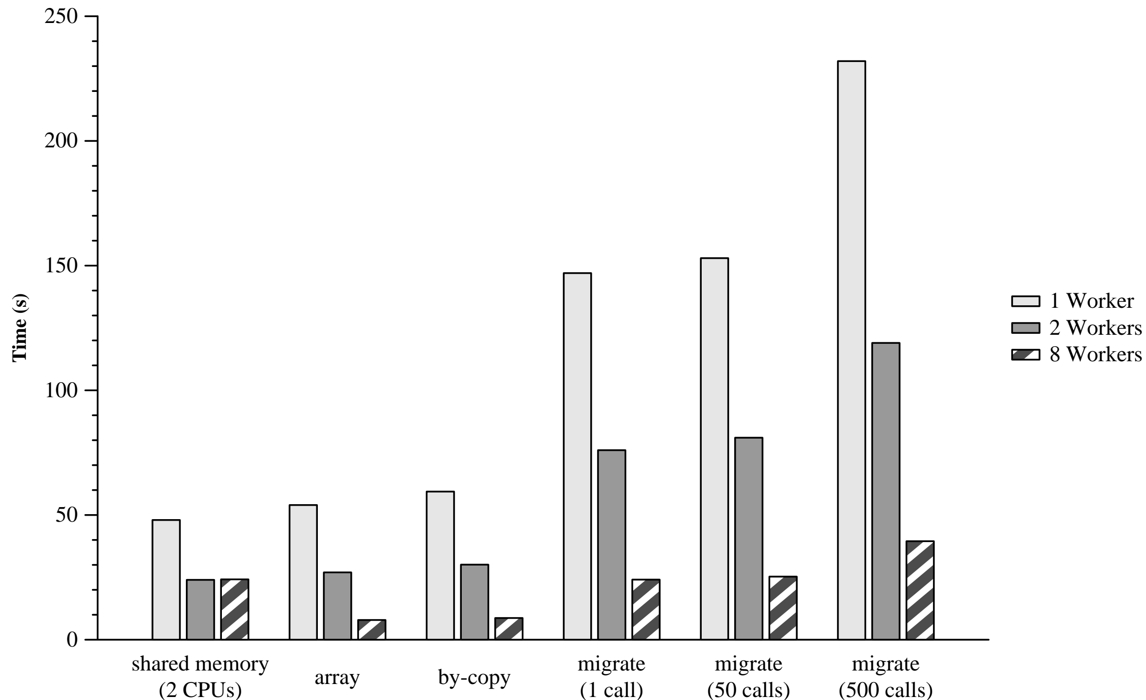

Figure 4.3: Absolute Performance of RC5

We did also run a version where the arrays are allocated statically on the master machine and accessed remotely by the workers. This version suffered prohibitive performance, though, being several thousand times slower than all other versions. The first, second and fourth version of the program above could be created simply by changing the configuration of the indefinite byte array object in the object graph. The third version is not currently expressible by Pangaea's configuration mechanism, but setting an object to "pass-by-move" would be a simple and worthwhile extension.

Some of the absolute execution times of the programs are shown graphically in figure 4.3. The centralized, shared-memory version naturally shows the best performance for up to two workers, but does of course not obtain any further speedup for more workers. The distributed *array* version is however only marginally slower for a single worker (by 12.5%), and outperforms the fastest centralized version for three workers and more. For eight workers (shown above), it takes 33% of the shared-memory/two-worker version's execution time, and for 32 workers, the execution time was 4.8 seconds, or 20% of the best centralized version (see below for the actual speedup graphs). This is of course due to the almost complete parallelism in this application.

It is interesting to note that the *by-copy* version, where the arrays are wrapped into objects that are then passed by-copy, is also not much slower than the "real" array version (again by only an additional 10%). This shows that Pangaea's array transformation is indeed not very expensive, which is of course due to the fact that array accesses are not very efficient in Java to begin with, involving a null pointer check and a bounds check. The additional cost of a local method invocation is therefore almost negligible.

If, due to the lack of static analysis such as factory method detection, the migration subsystem is put in charge of the arrays, a significant slowdown of about a factor of three is incurred however. In the *migrate* versions, the arrays are wrapped into objects, and calls to these objects are tracked by corresponding *Watcher* objects. It is mostly the cost of this book-keeping that

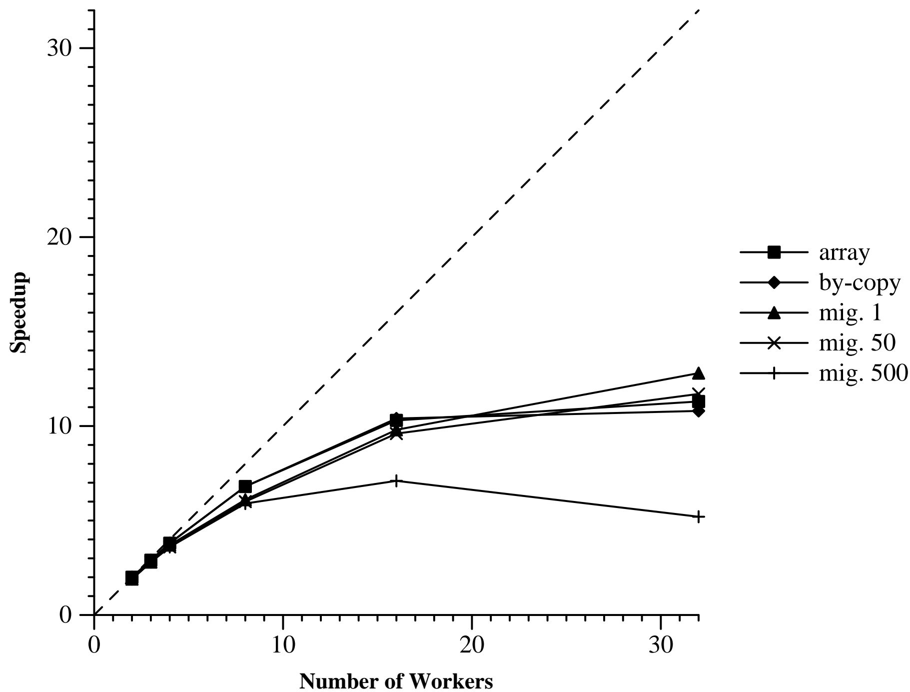

Figure 4.4: Relative Speedup of RC5

is responsible for the slowdown, as can be seen by varying the number of calls after which a migration is triggered. If a migration decision is made after each call, an array object is moved to its worker after the worker has made the ninth call to it. When the number of calls between migration decisions is increased, performance slowly decreases further, as the array objects stay on the manager's node for a longer time before they are moved. This is hardly noticeable for 50 calls, and becomes only significant for several hundred calls between migration decisions.

Figure 4.4 shows the speedup curves for the different distributed program versions, each of which is given relative to the execution of the corresponding program version with a single worker. The speedup is close to ideal for up to eight workers in each program version (efficiency better than 73%), and decreases as more workers are added. This is due to contention at the manager node, of which it is noteworthy that the book-keeping overhead of the *migrate* versions does not play a significant role except in the version with 500 calls between migration decisions. Only then does the high amount of book-keeping work on the manager node lead to earlier contention than with the other program versions.

The speedup for higher numbers of workers could of course be improved by making the work chunks larger, so that less communication with the central manager node is necessary.

4.2 A Ray Tracer

Our second case study is a simple ray tracer that was written without any relation to the Pangaea project[2] . We did not modify it in any significant way for the analysis, except in those places that will subsequently be explained. The size of the ray tracer is about 1,000 lines of code in 17 classes.

[2]The author is Ronald Veldema, VU Amsterdam; I would like to thank him for his permission to use this very instructive program.

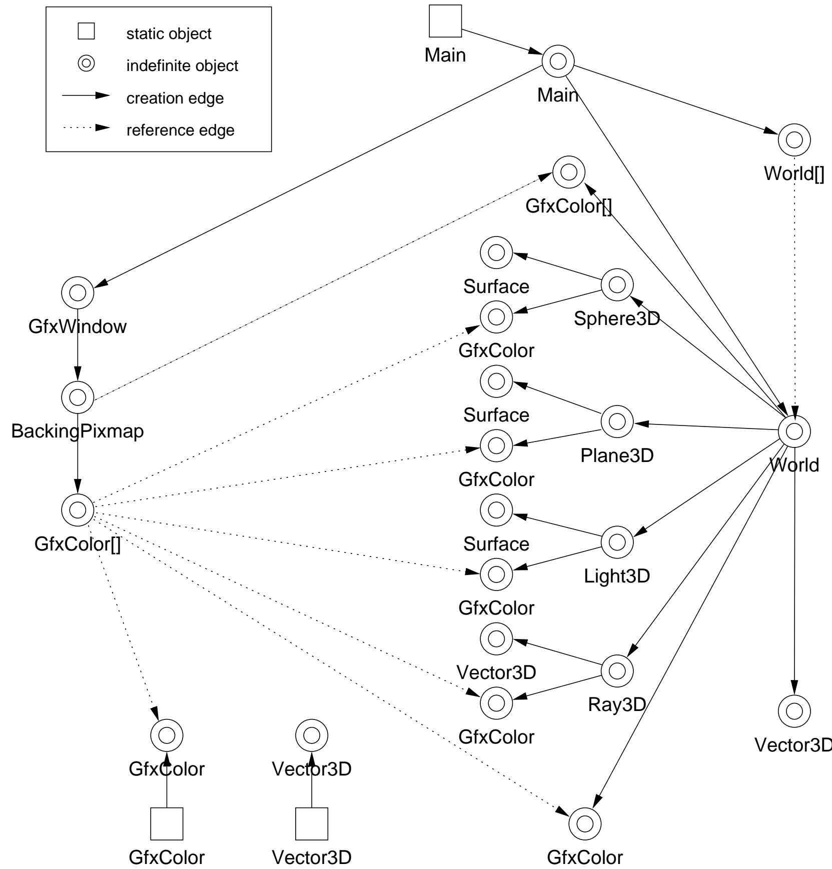

Figure 4.5: Object Graph of the Ray Tracer

The object graph of the ray tracer is shown in fig. 4.5 (we have slightly simplified it for presentation). Pangaea computes this graph in 13 seconds, of which 10 seconds are due to I/O and parsing (Sun Ultra-10, JDK 1.2, with JIT). The root node of the graph is the static part of the *Main* class, shown in the upper center. All objects created by *Main* are transitively reachable via the creation edges shown in the graph, excepting the static parts of the *GfxColor* and *Vector3D* classes shown lower left. The display engine, consisting of *GfxWindow*, *BackingPixmap*, and *GfxColor[]* objects is visible on the left; the tracing is performed by the *World* objects on the right hand side (each *World* is a thread), and their interior which represents the scenery (*Sphere3D*, *Plane3D*, etc.). The *GfxColor* objects represent pixel values (RGB); these are computed by the *World* objects and passed to the *BackingPixmap* in the form of individual rows, realized by the *GfxColor[]* array on the left. The *BackingPixmap* assembles these rows into the complete image, stored in another, private *GfxColor[]* array. (This array is also one-dimensional, with rows and columns being expressed by explicit index arithmetic.)

Apart from the three static objects, there are only indefinite objects in the graph. It is not typical for an object graph of this size to contain no concrete objects (which would represent exactly one instance). In this program, this is because internal objects are not created within constructors, but rather by explicit calls to initialization methods, which the algorithm, due to its flow-insensitiveness, cannot resolve. However, the fact that only indefinite objects appear in the graph does not hamper subsequent distribution analysis in any way.

The reference and usage structure within the graph is quite complicated, and we have omitted

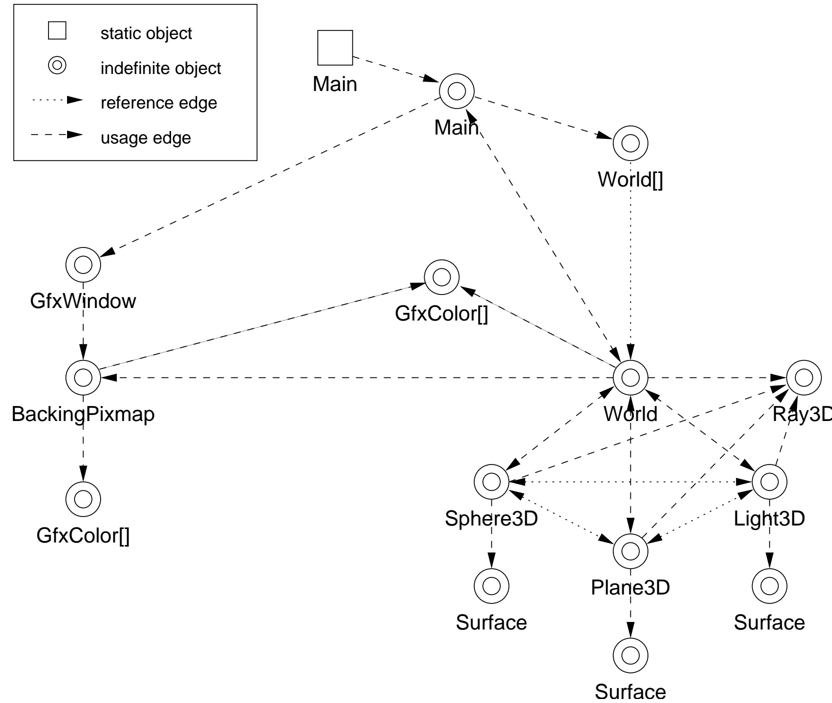

Figure 4.6: The Ray Tracer without Immutable Objects

most of the edges (there are actually about 250 reference edges and 150 usage edges in the graph). We have found that in general, it is useful to layout the graphs according to the creation edges; this gives a very good impression of the hierarchical structure of a program. Pangaea's user interface then allows to selectively display reference and usage edges connected to certain nodes (e.g. the reference edges emanating from the *GfxColor[]* array on the left).

Distribution Analysis

It is natural that the individual *World* objects, which are in fact thread performing the ray-tracing, should be placed onto distinct processors to achieve maximum parallelism. The display engine, comprising the objects on the left hand side of the graph, will likely have to be placed onto a distinct machine at which the display device is located. This distribution strategy is established fully automatically by Pangaea's graph assignment algorithm that we described in section 3.3, except for the fact not only the *World* threads, but also the display engine is placed onto a distinct abstract partition, since the *GfxWindow* object is itself a thread. This would have resulted in placing the display engine onto an arbitrary node at run-time, whichever the round-robin scheduler would have used. We corrected this by reassigning the *GfxWindow*, *BackingPixmap*, and the left *GfxColor[]* array to the root partition.

The excessive reference structure within the object graph does however not permit an efficient distribution this way. The *GfxColor* objects in particular, which represent individual pixels, would be allocated statically on the *World* nodes that created them, and thus would have had to be accessed remotely by the display engine. Object immutability can be used to remedy this situation, allowing the *GfxColor* objects to be passed to the display engine by value.

However, neither the *GfxColor* class, nor another obvious candidate, *Vector3D*, where immutable in the program as it was originally written. Both of these classes define containers for triples of integers with some additional arithmetic functions provided. They were not originally immutable because (a) the member fields were public and being read directly by the rest of the

program, and (b) arithmetic was often carried out *in situ*, i.e. an operation *v1.add(v2)* would add the values of *v2* to those of *v1* and return the modified object *v1* to the caller. By tightening the field protections, providing access methods, and performing arithmetic by creating new result objects and returning these, the two classes were made immutable with very few changes throughout the program. As the objects of these classes are now completely irrelevant for distribution purposes (they can be handled like values), they have been purged from the object graph in fig. 4.6. (In this graph, all creation edges are omitted, while usage and reference edges are complete — each usage edge implies a reference edge.)

In the centralized case, making the objects immutable did cause a slight performance penalty of about 5% execution time. In the distributed case, however, this change is what turns prohibitive performance into a workable behaviour (we were not able to get the version with global *GfxColor* objects to work due to RMI being overloaded and eventually terminating). The modification of the program to exploit immutability was of course done manually. This suggests that programmers need to be aware of the benefits of immutable objects for distribution, so that they write immutable classes whenever possible. It would also be helpful to have compiler or even language support to help programmers in ensuring this important property.

The remaining objects in the graph in figure 4.6 indicate how the display pixels are passed to the *BackingPixmap* row-wise, contained in *GfxColor[]* arrays. These arrays are created in what an algorithm like cJVM's would classify as a factory method (see the previous case study and page 47). This means that it can be proved statically that the creating *World* object does no longer maintain a reference to an array once it is passed to the display engine. Although Pangaea cannot yet detect factory methods, we exploited this property manually by deassigning the *GfxColor[]* array in the graph, causing it to be passed by value (along with the contained, immutable pixel objects).

Performance

We executed the ray tracer using Doorastha on the DAS-2 supercomputer in Amsterdam, under identical conditions as the RC5 program in the previous case study. The example scenery was a 128 by 128 picture containing several reflective spheres over a partially reflective, checkered floor. Each *World* object computed a different horizontal stretch of the image, the height of which was obtained by dividing the height of the image by the number of nodes.

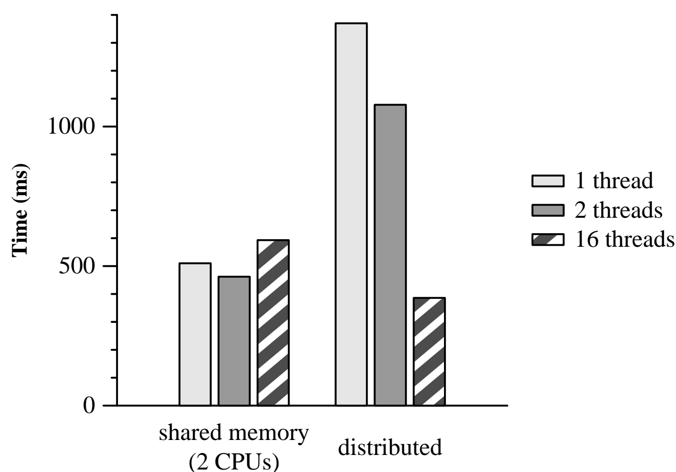

Figure 4.7: Absolute Performance of the Ray Tracer

Despite the many optimizations that we performed, the ray tracer showed only poor parallel performance (absolute execution times are shown in figure 4.7, relative speedup in figure 4.8).

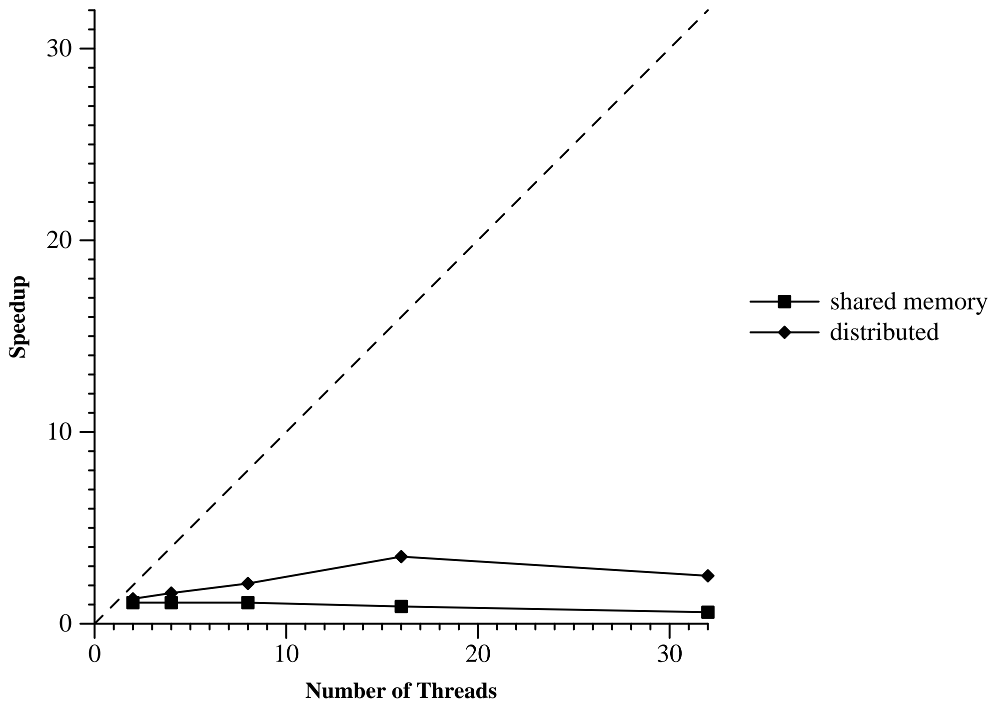

Figure 4.8: Relative Speedup of the Ray Tracer

The distributed program is slower than the shared-memory version by a factor of 2.7 for a single tracing thread, and only slowly improves as more nodes are added. Only with as many as 16 nodes does it begin to outperform the shared-memory version; the speedup however reaches its limits quickly after that, as shown in figure 4.8. (For some reason that we did not investigate further, the shared-memory version cannot capitalize on the dual-processor architecture either.)

The reason for the poor performance seems to be that — caused by the low execution time of barely a second — contention at the display engine occurs. (If this weren't the case, at least the relative speedup would be higher.) There is little that can be done about this at the level of *distribution technology*, without modifying the code of the program. One idea would be to group the individual rows in *batches* as they are delivered to the display engine. This could be done automatically by the distribution platform, or programmatically within the ray tracer itself, which would be an example for a design that needs to be adapted because of distribution considerations.

4.3 A Chess Opening Database

Figure 4.9 shows the user interface of a graphical database for chess openings. The program is intended to help chess players familiarize themselves with various openings. On the graphical chess board, the user can make arbitrary moves; the program looks these moves up in a database and displays the name of the corresponding opening, and possibly a commentary on the move. The database is implemented as a simple text file. Note that the program itself cannot *play* chess, it can only distinguish legal from illegal moves, and is simply an elaborated graphical interface to a database.

Pangaea allows us to convert this program into a distributed client/server application, where the user interface is displayed on the client machine, and the database resides, possibly separated by large physical distance, on a server. Unlike the concurrent programs that we studied in the

Figure 4.9: A Graphical Database for Chess Openings

previous sections, the goal in distributing this program is to keep the *penalty* of distribution as low as possible, in order to satisfy the external constraint.

Program Analysis

The program has about 2,500 LOC, its object graph is computed by Pangaea in 52 seconds (see page 72 for more details). The object graph is shown in figure 4.10. Unlike the previous graphs, we have greatly simplified this graph for presentation (omitting all immutable objects, for example).

The graph shows the graphical user interface on the left; the objects that implement the database are on the right. There are actually two separate *Board* objects: one is used as the application model for the graphical chess board on the screen; the other is used internally by the *Parser* to interpret algebraic chess move notation found in the text file. Some interesting properties can be shown regarding these two objects:

- The *Position* and *MoveList* objects used internally by the two *Board* objects are indefinite, because they are not created initially. However, despite the uncertainty about their actual numbers at run-time, the graph makes it clear that each *Board* has its own private objects of those types, and does not pass them to the outside.

- The left *Board* object communicates heavily with the user interface objects on the far left side. These interactions are realized through the Subject-Observer pattern, i.e. the *Board* object is a subclass of a *Subject* class that stores a list of *Observer* objects, which are updated on request. Of course, the *right Board* object also has such a list. However,

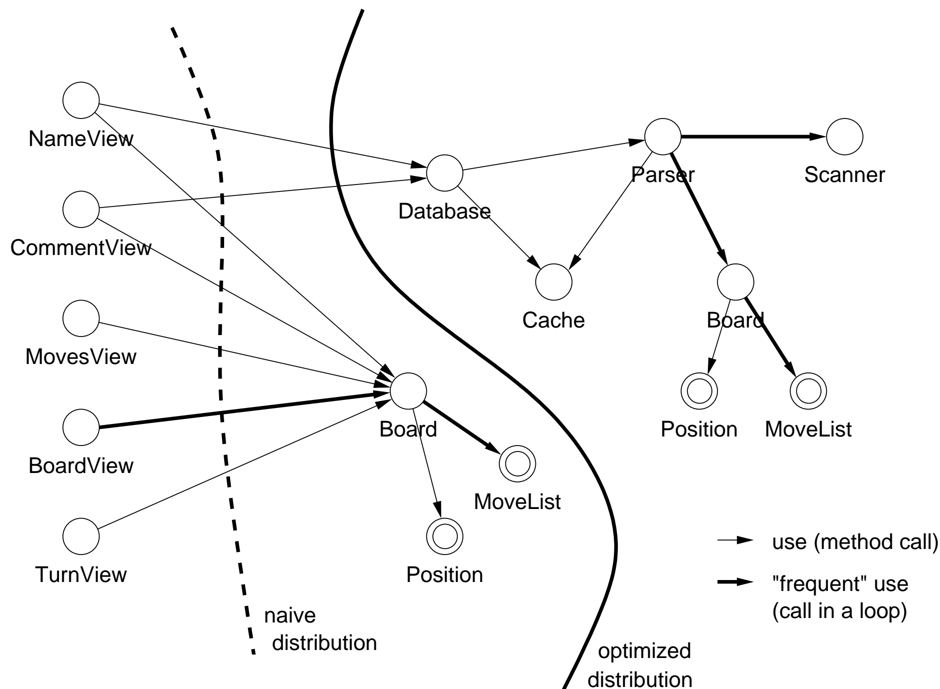

Figure 4.10: Object Graph for the Chess Opening Database

reference propagation shows that none of the *Observer* objects is ever registered with the right *Board* object, and therefore it cannot invoke methods of them at run-time.

Distribution Analysis

A naïve distribution where only the user interface objects are allocated on the client is obviously suboptimal for this program, because these objects communicate heavily with the left *Board* object that serves as the model for the GUI. For example, the entire chess board needs to be redrawn after each move, which involves 64 separate calls to the model to retrieve the position. An optimized distribution can be found by partitioning the graph so that the number of edges crossing the distribution boundary is minimized. As shown in figure 4.10, this means that the model objects are allocated on the client as well, and only the *Database* object needs to be invoked remotely.

Performance

We created both the naïve and the optimized distribution by manually assigning the objects to partitions. The JavaParty platform was used as the backend.[3] No object migration had to be used, since the program parts communicate via immutable objects. We executed the centralized and both distributed versions using a low-end Pentium 90 laptop as the client, and a Sun Sparc 10 as the server. Both a 10 MBit/s Ethernet network and a 28.8 kB/s modem were used to connect the machines, the latter being typical for connection speeds for normal Internet users. In the centralized version, both the user interface and the database were executed on the Sparc 10.

[3]Due to a problem in the most recent JavaParty version, we could not use the program that was generated by Pangaea to produce the numbers found in this section. Shown here are results for an equivalent program that was manually distributed using an earlier version of JavaParty.

A number of chess moves were then made on the user interface, and we measured the time until each move had been processed, involving the redrawing of the chess board and the display of the information retrieved from the database. This indicates the interactive response time of the application. The results are shown in figure 4.11 for the Ethernet connection, and in figure 4.12 for the modem. The light part of each bar indicates the time to the first visual response of the program, while the dark part is the added time until the interaction had completely been processed.

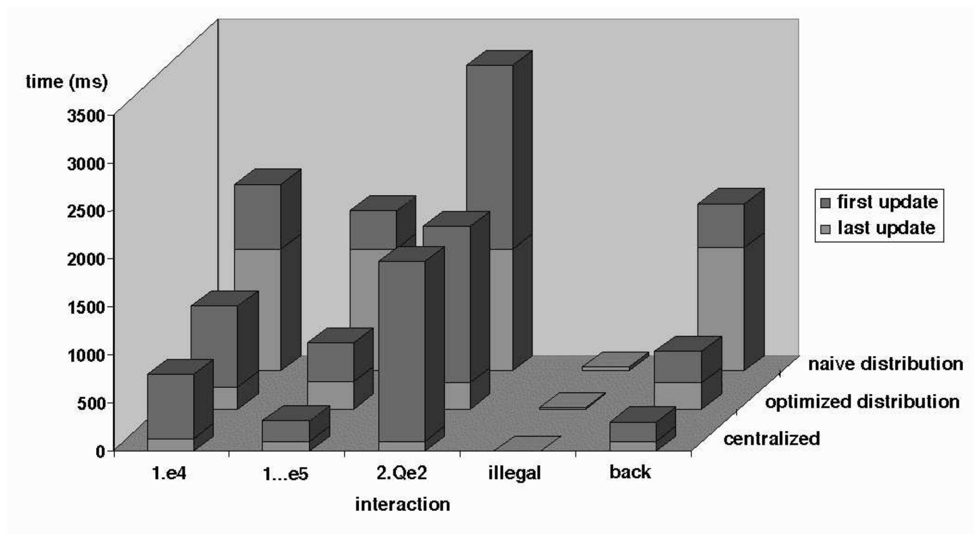

Figure 4.11: Performance of the Chess Program via Ethernet (10 MBit/s)

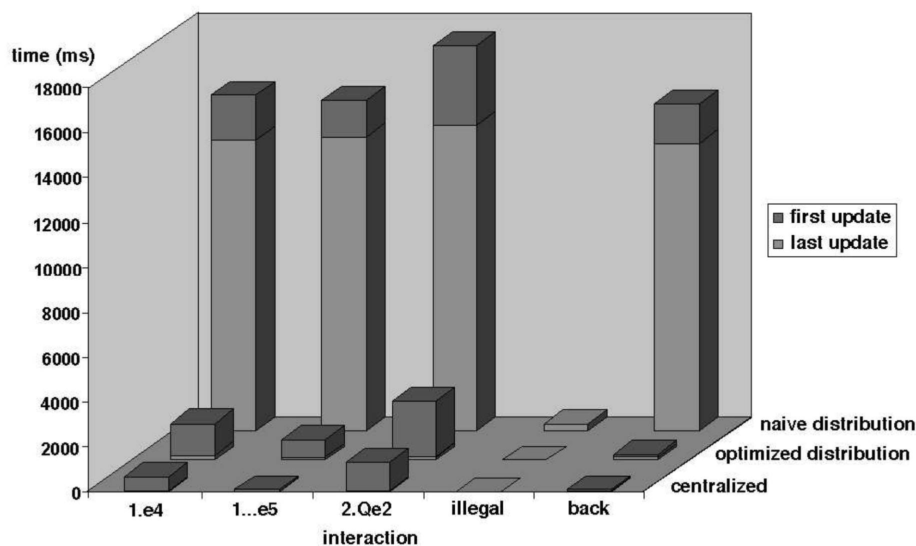

Figure 4.12: Performance of the Chess Program via the Modem (28.8 kB/s)

The graphs show that the response time of the naïve distribution is noticeably higher than that of the optimized distribution, which hardly exceeds that of the centralized version. On the fast network, the naïvely distributed program responds at least inconveniently slow to the user, while on the slow connection, its response time is simply unacceptable.