Exploiting Object Locality in JavaParty, a Distributed Computing Environment for Workstation Clusters

Bernhard Haumacher and Michael Philippsen

University of Karlsruhe, Germany
hauma@ira.uka.de and phlipp@ira.uka.de
http://wwwipd.ira.uka.de/JavaParty/

Abstract. In a distributed programming environment with location transparency, fast access to remote resources is absolutely critical for efficient program execution - but it is not sufficient. Locality optimization will try to group objects according to their communication patterns and replace remote access by local access whenever possible. Locality optimization is based on the assumption that local access always is much faster than remote access.
Using conventional communication packages like RMI, this assumption is void. A remote object always needs an access layer around it to allow access to the object by means other than direct reference. Transparent access to a remote object also needs to overcome this access layer, even in a local access. Direct reference access can not be used, because of different parameter passing semantics.
In this paper we present techniques to provide transparent local access to potentially remote objects in the same magnitude as direct Java access.

1 Introduction

JavaParty provides a distributed Java virtual machine on top of regular Java virtual machines that execute on the nodes of a workstation cluster. The distributed nature of the environment is mostly hidden from the application by providing a shared object space across all participating virtual machines. To make objects accessible from other nodes, Java's remote method invocation (RMI) is used [7].

Instead of asking the programmer to insert many verbose RMI commands manually into his multi-threaded Java application to port is from a single workstation to a cluster, JavaParty allows him to declare classes to be *remote*. JavaParty then generates all the necessary RMI commands automatically [6].

1.1 Location transparency

In JavaParty remote objects are location transparent in a way that the remote access to an object is syntactically and semantically identical to a local access to that object. The meaning of a JavaParty program therefore does not depend on the relative location of its objects. Location transparency enables the separation of application design from object allocation and distribution decisions. In particular this opens up the possibility to decide about the location of created objects automatically with compile time or runtime support. JavaParty also supports migration of already allocated objects to different nodes without the need for application interaction.

Classes not explicitly declared remote are regular Java classes, called *local* classes below. Local objects are bound to the virtual machine they are created in, and can not be accessed from outside that virtual machine. If local objects are passed in method invocations to remote objects, the RMI method invocation semantics apply. In an RMI call local objects as arguments are shipped to the remote side and instantiated there as copies identical of the original argument. Transparent access to a remote object requires that this duplication occurs even if the caller is local to the invoked remote object, because the application may rely on that behavior.

Transparent access to a remote object therefore requires some intermediate layer between the caller and the target object to guaranty the same semantics no matter where the object is located. Using regular RMI transparency also can be achieved by always accessing the remote object through its RMI proxy object. But this incurs an huge overhead for a local access that is comparable to a remote access in performance, because RMI does not exploit object locality. This paper presents techniques that can be used to make a communication package location aware in a sense that local access can be performed transparently with similar performance as a regular Java method invocation.

1.2 Frequency of local accesses to remote objects in JavaParty applications

Remote objects are the building blocks of a parallel distributed JavaParty application. Normally a JavaParty application will create many more remote objects than nodes are actually available for execution, because the application becomes more flexible when adjusting it to different target platforms. Especially when porting an existing multi-threaded Java application to JavaParty, many more remote classes are likely to be declared, because objects of remote classes have more Java-like semantics referring to the distributed virtual machine: They are always passed by reference in method invocations and are visible from the whole application.

An application that uses many more remote objects than nodes are actually available for execution relies on locality optimization to ensure that most of the accesses are local. In such a well optimized application the majority of accesses to remote objects will be local. Therefore the resulting application is only expected to execute efficiently if local access is quasi as fast as a regular Java method invocation.

1.3 Related work

To execute a program efficiently in a distributed environment, a fast network and slim protocol layers are necessary prerequisites. We have already show how to achieve this in pure Java [5, 3] on a Myrinet network. There are other efforts to improve the performance of Java remote method invocation. Henri Bal's group at Amsterdam has published results on the compiler project *Manta* [8]. Manta has an efficient remote method invocation (35μs) on a cluster of workstations connected by Myrinet by compiling Java to native code. Krishnaswamy et al. [4] improve RMI performance by using object caching and using UDP instead of TCP.

These approaches concentrate on fast remote access to objects in Java. While location transparency in not the main issue in this area, no performance numbers for transparent local access to remote objects are available. In this paper we study the behavior of the KaRMI, our optimized implementation of remote method invocation [5], and describe strategies how to add efficient transparent local access to a system that already is optimized for fast remote access over a high-performance network.

There are other communication packages other than RMI available for Java. An alternative approach for parallel programming in Java is the usage of MPI. There exist several MPI bindings for Java [1, 2]. While message passing is well-suited for statically distributed numerical applications, it does not fit the object-oriented nature of Java very well. For location transparent environments like JavaParty the remote method invocation approach is preferable because constructing messages is only worth wile when the target is known to be remote. The work described in this paper aims at fast transparent access to objects especially if they are local.

1.4 Structure of the paper

Section 2 describes improvements to KaRMI for early locality detection of method calls. It also describes further optimizations that can only be performed at the highest protocol layer and are essential for an efficient local method invocation. Section 3 describes optimizations to JavaParty for removing additional overhead introduced by the JavaParty layer. Section 4 proves the effectiveness of the optimizations presented in this paper with benchmark results. Section 5 concludes.

2 Optimizations at the remote method invocation level

2.1 Structure of KaRMI

Figure 1 shows the standard access path from JavaParty application code through the three KaRMI layers to the remote server implementation object. The steps are labeled (c1)...(c4) on the client side and (s1)...(s3), on the server side with the network communication (n) in between. To understand the possible optimizations for local access (l1)...(l3) we describe briefly the responsibilities of

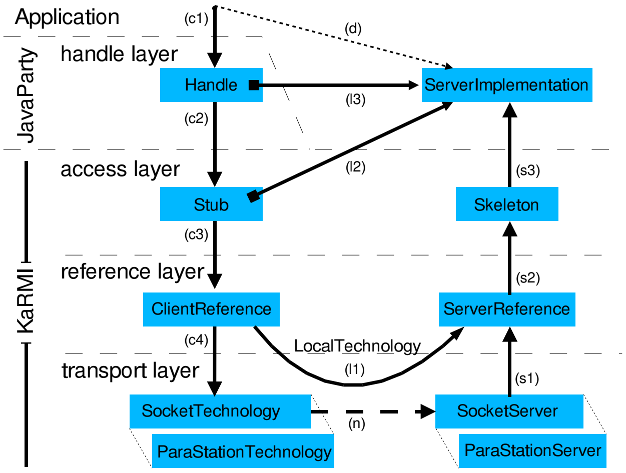

Fig. 1. KaRMI's Structure

the layers. Direct access to the server implementation (d) can not be used for transparent remote object access, because of different argument passing semantics as described in the introduction. For the moment we look at the system from the view of a regular KaRMI application by ignoring the section labeled "JavaParty" in the figure. Both regular RMI and KaRMI share the same high-level design, so most of the descriptions in the next paragraphs are valid for both implementations of remote method invocation.

The **access layer** of KaRMI provides the application with high level access to a remote object through a generated proxy object that has methods with the same signature as the server object. KaRMI is responsible for forwarding a method invocation performed on the proxy object to the server implementation and transmitting the result back to the caller. The proxy object, called "stub", is responsible for converting arguments passed to each of its methods into a representation that can be handled by the deeper generic layers of the package. On the server side the skeleton object performs the inverse transformation of the parameter list and result value by restoring the correct types of the parameters, invoking the requested method on the server implementation object, and converting the result back to a generic representation that is returned back to the caller.

The **reference layer** is responsible for uniquely addressing remote objects and dispatching incoming requests to the correct server side object. Remote objects are identified by so called object numbers that are assigned to a remote object at the time it is exported. The object number in combination with the Internet address of the machine and the port the object was exported on identifies the object in the distributed environment.

The **transport layer** is responsible for establishing and caching network connections to a remote machine using the appropriate network technology. The transport technology also defines the wire protocol that is used for the communication. At the moment there are two transport technologies implemented for KaRMI, one for regular TCP/IP sockets called `SocketTechnology` and one for the ParaStation [9] network layer over Myrinet communication hardware.

2.2 Detection of a local call in the access layer

Figure 1 shows the basic optimization (l1) for local calls. If the target of the call is detected to be local to the caller, the local transport technology is used. The technology for local transport is responsible for duplicating the parameter and result values to simulate the behavior that is expected form a standard remote method invocation. These call-by-copy semantics must be adhered to even if the call does not leave the local address space and therefore is not a real remote method invocation. The program may rely on the fact that remote calls have call-by-copy semantics and does not expect that modifications made to the argument of a remote method invocation get visible to the caller. In a location transparent environment based on KaRMI it is obvious that all calls to a remote object must have the same call semantics, because due to automatic object distribution, locality is not necessarily known to the programmer.

Locality detection of a remote call's target in the transport layer is too late: All layers of the packages are crossed back and forth unnecessarily. The access layer converts the argument list to a generic representation but the typed arguments are required for finally invoking the server method. The reference layer identifies the target of the call by object number, where a direct Java reference to the local server implementation object exists and would suffice. Finally the local transport technology performs costly object duplication on the generic method argument representation, where customized processing for most frequent parameter types would be faster.

Therefore the appropriate layer for detecting locality of a call is the access layer (l2) as far as a regular KaRMI application is concerned. Because the types of the method arguments are known at the access layer, there is the possibility of a specially optimization of argument duplication:

Basic types. Arguments of basic Java type need no explicit duplication, because they are always passed by copy even in a regular Java method invocation. These parameters can be passed directly in a local call to the server implementation without any additionall effort.

Remote references. Remote objects are represented at the client side by proxy objects called stubs. Stubs are remote references to server objects in the KaRMI framework. Such references can also be passed in a remote method invocation as method argument. While these stub objects can not be modified directly by the application, but only forward method invocations to the object they reference, the application does not care whether such a reference object is copied in a local invocation or not.

Remote servers. If a direct Java-reference to a remote server object is passed in a remote method invocation, it is replaced by a stub object referencing the original server object during the marshaling process of the call. Even if the remote method invocation is local, this conversion must be performed. But directly converting a remote server object to a stub referencing it is much less time consuming than doing the same operation during general purpose object duplication.

Immutable objects. Like remote stubs there is another category of objects that don't need duplication during a local call. All objects that are created once and that can no longer be modified are candidates for passing them directly in a local call to a remote method. Because the remote method cannot modify the state of the passed object, the application is unlikely to perceive the difference of a call-by-copy or a call-by-reference. Such immutable classes in the core Java library are strings and all object equivalents for basic types like `java.lang.Integer`.

Classes declared immutable. Since object duplication is expensive, it makes sense to enable the application designer to declare classes immutable and prevent the duplication of their objects in local calls. For this reason, we provide an additional interface `Immutable` that marks a class (and all its subclasses) to be immutable. The interface does not declare any methods, but is only used in the stub generator to decide whether a reference type that is not otherwise known is immutable and does not need duplication[1].

2.3 Redesign of the pure functional interface between KaRMI layers

The optimization presented in this section is an necessary prerequisite for the the one described in the next section, but also opens up the possibility of some interesting extensions to the whole system. Therefore it is worth describing it separately.

The main advantage of KaRMI over RMI are the simple pure functional interfaces between the layers. Such clean layering allows to easily add alternative transport technology implementations to support high performance network hardware. A pure functional interface means that the only interaction between two layers of the package during the execution of a remote method invocation is exactly one method call. This method call ships all necessary information that is required to perform the remote invocation to the next lower layer. The lowest layer is responsible for constructing the network packet that ultimately ships the call to the server side.

First we show, that such pure functional interface is inappropriate because it causes unnecessary object allocations during a remote invocation. We present a solution to the problem without violating the extensibility of the system to alternative transport technologies.

Using pure functional interfaces for the layers in KaRMI requires that each layer collects all necessary information and forwards it in a single call to the next

[1] We decided to have the programmer declare a class immutable instead of trying to detect this property by code analysis, because there are several cases where code analysis can not detect a class to be immutable, e.g. non-final classes.

deeper layer. Because the transport layer is responsible for constructing the byte representation of the information sent over the network, the higher layers must forward their information as Java objects to the next layer. This approach does not fit the access layer very well. The stub is responsible for dealing with the method signature and converting the arguments to a generic representation. The deeper layers of the system do no longer deal with the method signature but are only capable of forwarding remote calls requiring that the arguments are packed in a format that can simply be forwarded to the server side. On the server side the skeleton is responsible for extracting the arguments by restoring the concrete method signature and actually invoking the remote server method. In Java there exists no efficient representation for an arbitrary method signature, that can be used to represent the arguments of a method invocation. The Java 2 version of RMI uses an array of objects for the arguments and expects a single object as result. This representation requires the allocation of an array object for each method invocation and the wrapping of each argument with basic Java type into an object. On the server side the Java 2 version of RMI does not use a skeleton object, but invokes the method of the server implementation object through the reflection mechanism. The representation of the method arguments is inefficient because it requires multiple object allocations for simple method signatures. Using dynamic type inspection to invoke the implementation of the method on the server side causes additional overhead that can easily be avoided by generating a customized skeleton object. KaRMI did prevent the wrapping of basic types to objects by generating a parameter object class for each method signature that is used in a remote object. The parameter object is used as container object to tunnel the invocation parameters through the generic layers. This parameter object is equipped with custom marshaling and unmarshaling methods for efficient transmission over the network.

Obviously the approach using a parameter object is not the most efficient solution, because at least one additional object is allocated for each method invocation. No wrapping of basic types to objects is necessary, but all method parameters are first copied to instance variables of the parameter object and then marshaled to the network stream. This was necessary to fulfill the design criterion of a functional API between the layers described above.

To prevent the additional parameter conversion, the stub object must marshal the method arguments directly. To achieve this, we split the method invocation process in three phases. The first phase is responsible for establishing a method invocation context, in the second phase the method is invoked, and in the third phase the result of the method is returned. The last two phases are under the control of the stub using the invocation context. The KaRMI layers have only to be crossed in the first phase while the object addressing and the correct transport technology is detected. The invocation context that is the result of the first phase is part of the transport layer and encapsulates a network connection. Each transport technology installed in KaRMI provides its own invocation context implementation.

```
// --- Phase 1 ---
Connection c = ref.getContext( ... );

// --- Phase 2 ---
c.openSendCall();
  // marshal the method arguments to c
c.closeSendCall();

// --- Phase 3 ---
if ( c.openReceiveResult() ) {
  // unmarshal the method result from c
} else {
  // unmarshal an exception returned from c
}
c.closeReceiveResult();
```

Fig. 2. Phases of a remote method invocation

Figure 2 shows the actions performed in the three phases. After the context is established, the stub starts the method invocation with `openSendCall()`. After that call the context is ready to marshal the method arguments. The marshaling process is finished by calling `closeSendCall()`. `openReceiveResult()` requests the result of the remote method invocation. The method invocation may either complete normally by returning the expected result or return exceptionally by returning an exception object. Which case occurs is indicated by the result of `openReceiveResult()`. Finally the context is passed back to the system for reuse in the call to `closeReceiveResult()`.

This approach has two benefits over a pure functional design where simply one method of the next deeper layer is invoked. First the stub can directly marshal the method arguments with marshaling methods provided by the context object, so no additional conversion of the arguments takes place. This approach makes a remote method invocation with no additional object allocation possible, because the context object is cached and reused for following invocations. There is no need for wrapping basic types that are the most frequent method parameters.

The second benefit is the fact that issuing the method invocation is separated from the receipt of the result of the method invocation. This separation is necessary for a general purpose implementation of asynchronous remote method invocations. RMI does not support asynchronous remote method invocations, because in Java all method invocations are synchronous and an asynchronous invocation does not fit into the concept of RMI where the server implementation object and the generated remote proxy object must implement a common interface. An asynchronous remote invocation must necessarily have a signature different from the server implementation method, because the remote method can no longer be represented by a single method in the proxy. It must be split into issuing the call and receiving the result. The proxy is generated after the application is compiled, so the only way of accessing the proxy from the pre-compiled application is an interface that is known in advance. If the remote method invocation package is used in the context of JavaParty, this limitation no longer holds. While it is inconvenient and error-prone if the optimization of hiding network latencies is left to the programmer, asynchronous remote method invocations make latency hiding possible that may be performed automatically by the compiler.

3 Optimizations at the JavaParty layer

So far we have presented optimizations that only involve the remote method invocation package KaRMI. Besides JavaParty, all applications using KaRMI as replacement for RMI benefit from the optimization of early locality detection in the access layer (l2) and the smart argument copying strategy introduced in section 2.2. As shown in Figure 1, JavaParty adds an additional layer to the system to provide *transparent* remote objects and object migration support. This is accomplished through adding a handle layer that also uses a proxy object to access the remote server implementation, but also is aware of migration.

If the object that is the target of the method call is on a remote machine, the additional layer does not degrade performance significantly, because there is not much computation performed in the handle and the network access clearly dominates the overhead of the additional layer.

In case of a local call with the shortcut (l2) in KaRMI, the additional layer slows down the invocation significantly because instead of two method invocations (from application code to the stub object in the access layer and from there to the server implementation object) that are necessary for a local KaRMI call so far, three invocations are necessary for a local JavaParty invocation. (that corresponds to a slowdown of 50%).

As shown in Figure 1, the logical consequence is the shortcut invocation (l3) that invokes the server implementation method directly from the JavaParty handle object in case the call is detected to be local. Because this requires knowledge of the internal KaRMI structure within JavaParty, we decided to inline the code of the KaRMI stub objects directly into the JavaParty handle by integrating the stub generator into the JavaParty compiler. The JavaParty compiler now can generate smart handle objects that are aware of object migration (the JavaParty functionality) and can decide about locality of the referenced object by either invoking the server method directly or accessing the reference layer of KaRMI to perform a remote invocation (the KaRMI functionality).

4 Benchmark results

Figure 3 shows benchmark results for calls to empty methods with different parameter and result types. The first method studied is a `void` method that takes no parameters, the second is a method that takes no parameters but returns an integer as result. The third and fourth take one and two int-parameters respectively. The fifth method takes a parameter of immutable type as described in section 2.2, while the sixth one takes a parameter that can be fast marshaled using the `uka.transport` package. Note that the time consumed by one method call is given in microseconds with a logarithmic scale.

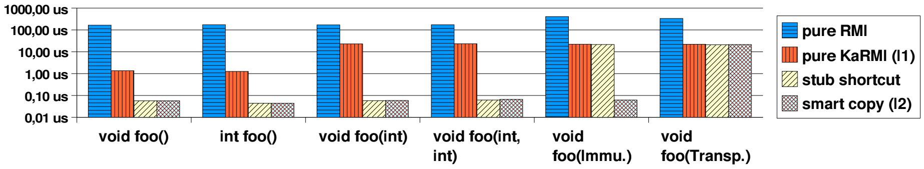

Fig. 3. KaRMI benchmark results

The benchmark illustrated in Figure 3 studies bare local RMI/KaRMI communication. Therefore the methods are called on a stub object that is located in the same virtual machine as the server. The first two bars of each group are for reference only. The first bar (with horizontal stripes) describes the time duration of a regular local Java-RMI call. The second bar (with vertical stripes) shows the performance of a regular local KaRMI call without the improvements described in this paper. As you can see, KaRMI already performs much better than RMI. For methods with no parameters and void or basic return type, the performance is better by two orders of magnitude. Even for methods with arguments (of basic or object type), KaRMI outperforms RMI by a large factor (7.4 up to 18.4), but the KaRMI performance is still far from optimal.

The last two bars of each group (slanted and double striped) show for each method benchmark the performance of a local call with the optimizations of KaRMI described in section 2. Even for the `void` method an additional acceleration factor of 24 is achieved by local call detection at stub level (stubShortcut). Where KaRMI does only save little in comparison to RMI, the locality detection at stub level (stubShortcut) and avoiding of argument copying (smartCopy) does the rest. In all three cases, where costly argument copying can be avoided the same good absolute performance can be achieved. Where the duplication of the argument is necessary to preserve the semantics of a remote call, the effect of the optimization is not perceivable because of the huge overhead for object duplication.

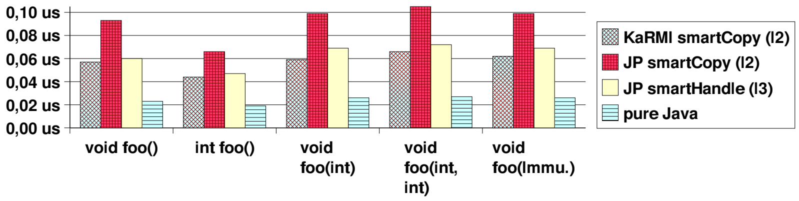

Fig. 4. JavaParty benchmark results

Results in Figure 4 compare the achieved performance of optimized KaRMI (KaRMI smartCopy) with a version of JavaParty that simply uses that KaRMI

μs per call	pure RMI	pure KaRMI	KaRMI stub-Shortcut	KaRMI smart-Copy	JP smart-Copy	JP smart-Handle	pure Java
void foo()	166.103	1.367	0.057	0.057	0.093	0.060	0.023
int foo()	172.970	1.260	0.044	0.044	0.066	0.047	0.019
void foo(int)	170.606	23.032	0.058	0.059	0.099	0.069	0.026
void foo(int, int)	172.253	23.265	0.062	0.066	0.105	0.072	0.027
void foo(Immu.)	409.866	22.221	21.595	0.062	0.099	0.069	0.026
void foo(Transp.)	334.720	22.071	20.958	21.469	22.004	22.192	0.024

Table 1. Combined view of KaRMI and JavaParty results

version (JP smartCopy) and JavaParty with the additional optimization described in section 3 (JP smartHandle). For reference, the performance of a regular Java method call is depicted in the last row (pure Java). Note that the scale of this figure is linear in contrast to Figure 3. The first bar (KaRMI smartCopy) is identical to the last bar in Figure 3 but rescaled.

You can see that inlining the stub functionality into the JavaParty handle layer and using the shortcut (l3) for local calls as described in section 3 is able to almost completely avoid additional JavaParty overhead.

Comparing the time, a local JavaParty method invocation takes, and the time for a regular Java method call, proves the resulting JavaParty system to be efficient: A local JavaParty method invocation (JP smartHandle) takes only a factor of $2.5 \ldots 2.7$ longer than a regular Java method invocation[2]. You would expect a slowdown factor with that magnitude, because instead of a single method invocation in regular Java two invocations are necessary due to the proxy object. Additionally a test for locality and a type conversion of the server reference is necessary before the target method can be invoked. Table 1 presents a combined view of the detailed benchmark results for KaRMI and JavaParty.

5 Conclusion

In this paper, we presented techniques to realize local access to transparent remote JavaParty objects within the same order of magnitude as a regular Java method invocation. The results are not restricted to JavaParty, because most of the optimizations were performed in the KaRMI remote method invocation package. The locality aware KaRMI package can be used as replacement for RMI, the standard Java class library for remote method calls. Further on we presented the fundamentals for extending KaRMI to asynchronous remote method calls.

References

1. Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. Object serialization for marshaling data in a java interface to MPI. *Concurrency: Practice and Experience*, 12(7):539–553, 2000.

[2] If copying of method arguments can be avoided.

2. Vladimir Getov, Paul A. Gray, and Vaidy S. Sunderam. Aspects of portability and distributed execution for JNI-wrapped message passing libraries. *Concurrency: Practice and Experience*, 12(11):1039–1050, 2000.
3. Bernhard Haumacher and Michael Philippsen. More efficient object serialization. In *Parallel and Distributed Processing*, number 1586 in Lecture Notes in Computer Science, Puerto Rico, April 12, 1999. Springer Verlag.
4. Vijaykumar Krishnaswamy, Dan Walther, Sumeer Bhola, Ethendranath Bommaiah, George Riley, Brad Topol, and Mustaque Ahamad. Efficient implementations of Java Remote Method Invocation (RMI). In *Proc. of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'98)*, 1998.
5. Michael Philippsen, Bernhard Haumacher, and Christian Nester. More efficient serialization and RMI for Java. *Concurrency: Practice and Experience*, 12(7):495–518, May 2000.
6. Michael Philippsen and Matthias Zenger. JavaParty: Transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, November 1997.
7. Sun Microsystems Inc., Mountain View, CA. *Java Remote Method Invocation Specification*, October 1998. ftp://ftp.javasoft.com/docs/jdk1.2/rmi-spec-JDK1.2.pdf.
8. Ronald Veldema, Rob van Nieuwport, Jason Maassen, Henri E. Bal, and Aske Plaat. Efficient remote method invocation. Technical Report IR-450, Vrije Universiteit, Amsterdam, The Netherlands, September 1998.
9. Thomas M. Warschko, Joachim M. Blum, and Walter F. Tichy. The ParaStation project: Using workstations as building blocks for parallel computing. *Journal of Information Sciences*, 106(3–4):277–292, May 1998.