Native and Generic Parallel Programming Environments on a Transputer and a PowerPC Platform

Hoekstra, A.G.; Sloot, P.M.A.; van der Linden, F.; van Muiswinkel, M.; Vesseur, J.J.J.; Hertzberger, L.O.

Published in:
Concurrency Practice and Experience

(34)

CONCURRENCY: PRACTICE AND EXPERIENCE, VOL. 8(1), 19–46 (JAN–FEB 1996)

Native and generic parallel programming environments on a transputer and a PowerPC platform

A.G. HOEKSTRA, P.M.A. SLOOT, F. VAN DER LINDEN, M. VAN MUISWINKEL, J.J.J. VESSEUR AND L.O. HERTZBERGER

Parallel Scientific Computing & Simulation Group
Computer Systems Department
Faculty of Mathematics and Computer Science
University of Amsterdam
Kruislaan 409
1098 SJ Amsterdam, The Netherlands

SUMMARY

Genericity of parallel programming environments, enabling development of portable parallel programs, is expected to result in performance penalties. Furthermore, programmability and tool support of programming environments are important issues if a choice between programming environments has to be made. In this paper we propose a methodology to compare native and generic parallel programming environments, taking into account such competing issues as portability and performance. As a case study, this paper compares the Iserver–Occam, Parix, Express and PVM parallel programming environments on a 512-node Parsytec GCel. Furthermore, we apply our methodology to compare Parix and PVM on a new architecture, a 32-node Parsytec PowerXplorer, which is based on the PowerPC chip. In our approach we start with a representative application and isolate the basic (environment)-dependent building blocks. These basic building blocks, which depend on floating-point performance and communication capabilities of the environments, are analysed independently. We have measured point-to-point communication times, global communication times and floating-point performance. All information is combined into a time complexity analysis, allowing comparison of the environments on different degrees of functionality. Together with demands for portability of the code and development time (i.e. programmability), an overall judgement of the environments is given.

1. INTRODUCTION

Real success of massively parallel processing critically depends on programmability of parallel computers and on portability of the parallel programs. We are led to believe that 'parallel computing has come to age'. Although it is safe to say that parallel hardware has reached a convincing stage of maturity, both the programmability of the parallel hardware and the portability of parallel programs still pose serious problems to developers of parallel applications. Today, an application programmer is usually faced with a situation as drawn in Figure 1 (see also[1]). A parallel computing platform supports native environments, which allow low level programming, or allow a more abstract view of the hardware. Furthermore, generic environments, also available on other platforms, can be used. These environments can be grouped in order of decreasing hardware visibility and increasing portability. Of course, one expects that the price to be paid for portability is a decrease of control of the hardware and associated degradation of performance.

In this paper we address the question of how to compare (native and generic) parallel programming environments, taking into account issues such as performance, portability

CCC 1040–3108/96/010019–28

Received 7 July 1994
Revised 14 December 1994

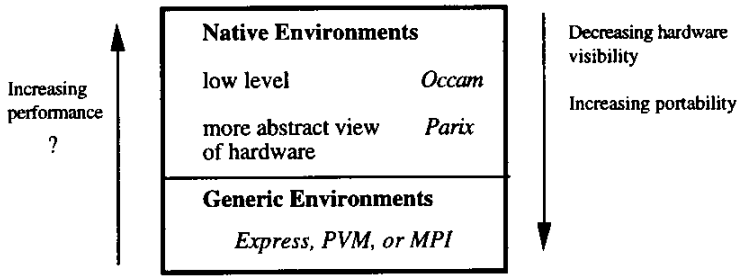


Figure 1. A typical situation encountered by application programmers of parallel systems

and availability of tools. We propose a methodology and apply it to a case study of native and generic environments on a transputer platform and on a PowerPC platform.

A typical example of native and generic environments is provided by the developments in transputer-based parallel processing. The first generation of these parallel systems consisted of transputers hardwired into grids, and they had to be programmed in the transputer's native language, Occam. The programmer had to know all the details of the parallel hardware, routing of messages had to be done explicitly, and the topology of the network was fixed. The next generation allowed software-reconfigurable topologies and programming of the system in the standard languages C and Fortran 77, extended with message-passing primitives. Furthermore, programming environments, like, for example, CS-Tools[2], allowed the sending of messages between processes, not necessarily located on adjacent processors: the routing was done implicitly by the system. The next step should have been systems based on the new T9000 transputer. The delay in production of this chip forced manufacturers to turn to other chips, or move back to T805 transputers and emulate the T9000 virtual channel routing[3] in software. This resulted in, for example, the GCel series of Parsytec[4], with its programming environment Parix[5]. Here, concurrent processes communicate via virtual channels and the machine can be configured into virtual topologies.

Despite these efforts, portability of parallel programs, developed in native transputer environments, to other parallel computing systems is rather poor. Fortunately, parallel programming environments have been developed which are supported on a large number of different parallel computers. These environments carry the potential of true source level portability of parallel programs between very different types of parallel systems, including clusters of workstations and heterogeneous systems. Two very popular environments are now also available on transputer-based systems. Firstly, the Express[6] system of Parasoft, which is available on, for example, networks of workstations, or the Intel Paragon, is also available on transputer systems. Secondly, the PVM environment[7], which has become the *de facto* parallel programming environment, has been ported to the Parsytec GCel.

Summarising, nowadays an application engineer, developing parallel programs or porting large sequential codes to massively parallel systems, has to choose between many different programming environments (for an overview, see[8]). This choice will be based

on competing issues such as development time, portability, availability of debugging and profiling tools, ease of use, and last but not least, performance. Many research groups have therefore compared parallel programming environments[9–13]. The majority of such comparisons, however, concentrate around clusters of workstations, and have not analysed the behaviour of programming environments such as PVM or Express on true massively parallel machines. Furthermore, in many cases only the communication primitives of the environments were examined, without assessing the implications of the results for real parallel applications.

The goal of this work is to propose a strategy to compare parallel programming environments, and to apply this to native and generic programming environments running on a large massively parallel system. We compare two native parallel programming environments, Iserver-Occam and Parix, with two generic environments, Express and PVM, by examining the behaviour of a representative parallel application implemented in these environments. These experiments are executed on a 512-node Parsytec GCel. As a case study we have implemented an application from physics, i.e. elastic light scattering simulations using the coupled dipole method[14–16] on the Parsytec GCel. This application has the following characteristics:

- It is a real application. This means that the application is actually used for simulations.
- The time complexity of the program is predictable. The execution time of the program can be expressed in terms of problem size, number of processors and a small set of basic system parameters (see, for example, the approach as described in[17]). This allows a first comparison of the environments by measuring this very limited set of parameters.
- It contains global communication routines. Global communication requires routing of messages to all processors. Explicit coding of this in Occam, for example, is an extensive programming effort, which is not necessary in environments like PVM.
- The implementation does not exhibit load imbalance. Load balancing is a research area by itself, and would obscure our current experiment. The Parsytec GCel is a monolithic platform, i.e. all processors have the same capabilities, unlike networks of workstations, for instance. Therefore, the only possible source of load imbalance in our experiment would be the program itself.

The coupled dipole application is an instance of a large class of applications where matrix–vector products are the main computational effort. Many other applications using iterative numerical solvers also fall in this class of applications.

We analyse the behaviour of the parallel coupled dipole method in the four environments by analysing both basic and global communication routines, floating-point performance, and actual execution times of the parallel program as a function of problem size and number of processors. We investigate if the basic measurements can predict the run time of the application, and if such basic measurements can be used as a heuristic to assess the merits of a programming environment. In this way we can judge the trade-off which exists between native environments, usually offering a better performance at the price of extensive programming effort, and generic environments which allow development of more portable programs. Finally, we apply our methodology to assess the merits of Parix and PVM on a very recent architecture, the Parsytec PowerXplorer.

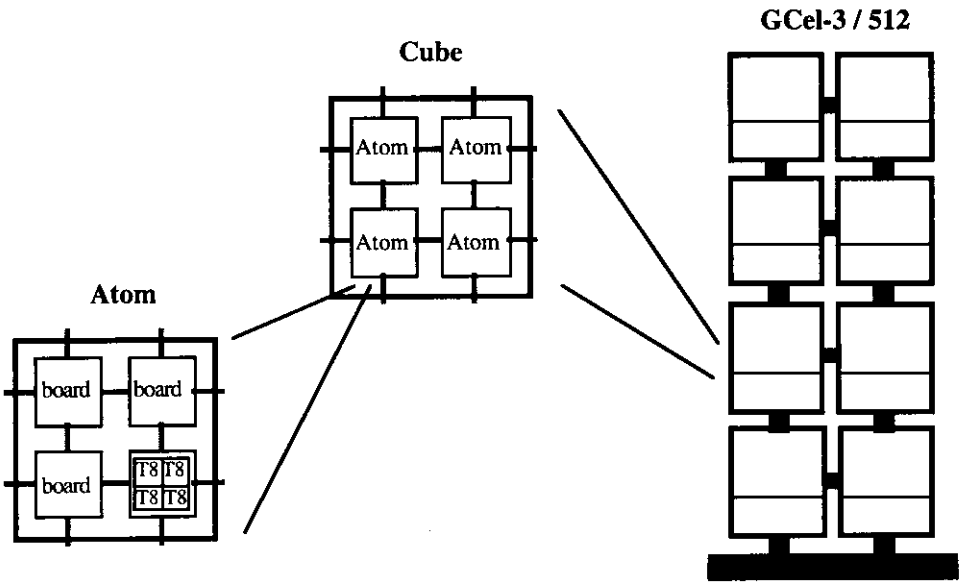


Figure 2. Schematic drawing of architecture of Parsytec GCel-3/512 installation

2. DESCRIPTION OF THE HARDWARE AND THE SOFTWARE ENVIRONMENTS

2.1. The Parsytec GCel

The Parsytec GCel-3/512, which was installed at IC^{3}A[1] in Amsterdam in January 1993, consists of 512 T805 transputers, running at 30 MHz, with 4 Mbyte external RAM and 4 kbyte on-chip RAM each. The transputers are physically connected into a 16×32 two-dimensional grid. The architecture of the machine is drawn schematically in Figure 2.

The basic logical unit of the GCel is an 'atom', constituting a total of 16 transputers. An atom consists of four boards each equipped with four transputers. The basic building block of a GCel is a 'cube', containing four atoms, i.e. 64 transputers. GCel installations consist of multiple cubes, in our case eight.

All atoms are arranged in a large number of overlapping partitions which users can request. In this way the user can obtain parts of the machine containing multiples of 16 transputers. Users access the machine via a front-end, a Sun Sparc 10 workstation.

2.1.1. The Parsytec PowerXplorer

Recently Parsytec has launched a new series of parallel computers which have the Motorola PowerPC as the compute engine. Transputers are used for communication between the nodes. In Amsterdam we have installed a 32-node Parsytec PowerXplorer. Each node consists of 1 PowerPC-601 running at 80 MHz, with 32 Mbyte RAM and 1 T805 transputer for communication. The nodes are connected in a 4×8 grid.

[1] IC^{3}A is the Interdisciplinary Center for Complex Computer Facilities, Amsterdam. For more information, e-mail: ic3a@fwi.uva.nl.

2.2. Parix

The standard programming environment for the PowerXplorer and the GCel is Parix[5], an acronym for parallel extensions to Unix. All experiments described here are performed under Parix release 1.2. Parix is a cross-development system. Applications are developed and compiled on the front-end, and subsequently downloaded to the nodes. The programming model is single program multiple data (SPMD)[5]; i.e. every processor contains the same main program. However, depending on the data allocated to a processor, different branches of the program can be executed.

The processors synchronise and exchange data by means of message-passing. Communication is performed through so-called virtual links, and can be synchronous and asynchronous. Every processor can define a virtual link to any other processor in the network. Parix takes care of routing the messages through the network. A set of virtual links between different processors can be grouped into virtual topologies. Users can define their own virtual topologies, or can use existing virtual topologies, such as rings, meshes or trees, by calling appropriate functions from the virtual topology library of Parix. These library functions guarantee an optimal mapping of the virtual topology on the actual physical two-dimensional network of the parallel system. Messages can also be exchanged without using virtual links: however, this mechanism is not very efficient. Furthermore, new releases of Parix will contain global communication routines, such as broadcasts or global sums. However, during development of the application these routines were not yet available, and are therefore not included in the experiments described in this paper.

Parix programs can be developed in ANSI-C or Fortran 77, and all standard Unix tools can be used. Communication between processors is performed through library calls. Input and output to the front-end is performed transparently through remote procedure calls to appropriate Unix system calls, and communication with the front-end is also possible through Unix sockets. Parix 1.2 is equipped with a performance analyser called Patop.

2.3. Iserver

The Iserver-Occam environment consists of the Occam toolset to develop Occam 2 programs and run them on, in our case, the Parsytec GCel. The transputer and Occam were developed together, and Occam can be considered the native language of the transputer. Occam allows maximal exploitation of the transputer hardware. Programs are developed and compiled (compiler version d7205) on the Unix front-end, and subsequently downloaded on the Parsytec. Mapping of transputer links to Occam channels, and configuration of the network, are carried out by a special configuration file, which is included in the Occam program.

Occam can only handle synchronous communication through Occam channels. If such a channel is mapped onto a transputer link, parallel processes running on neighbouring transputers can exchange data. To send data to an arbitrary processor the programmer has to route the data explicitly through the network.

2.4. Express

The Express system[6,18] is a product of Parasoft, and is based on the early work of Fox *et al.*[19]. Express is available on a wide range of platforms including Suns, Cray systems and

Meiko systems. Currently we have a β-version running on the Parsytec GCel[20]. Express consists of a set of libraries to describe the parallelism in the program. For instance, a communication library is present, offering primitives for sending messages between nodes, and high-level global communication routines such as broadcast and global data gathering. Furthermore, Express contains a parallel graphics system that offers a variety of graphical functions to all nodes in the system. Express supports C and Fortran77 programming, and contains a number of programming tools such as a parallel debugger and a graphical performance analyser.

Express offers two different working models: the host-node model and the so-called cubix model. In the host-node model one dedicated node, usually the front-end machine of the parallel system, starts and controls the computation on the parallel nodes, and all I/O operations have to be performed by the host. In the cubix model a dedicated host node does not exist, and all work is performed by the parallel nodes. Operating system services, that the parallel nodes may require, are transparently redirected to the front-end. The cubix model implies SPMD programming and is comparable with Parix' programming model. The coupled dipole program is implemented in the cubix model.

A cubix program is executed as follows. First, a partition is booted in the Parsytec GCel. Next, Express is initialised by loading the Express kernel and the routing tables on each transputer in the partition. Finally, the program is started by loading it on a user-specified number of processors inside the Express partition.

On the Parsytec version the user has no control over the mapping of the parallel processes into the partition. Furthermore, the physical location in the partition is unknown, only process identification numbers can be assessed. However, by using, for example, the exgridinit call, the parallel processes can order themselves into N-dimensional grids and communicate in, for instance, the left or north direction. This is comparable to the virtual topologies of Parix.

2.5. PVM

PVM (parallel virtual machine) is a system to support heterogeneous distributed computing[7,21]. A virtual machine may include heterogeneous computers, such as workstations, vector machines or monolithic massively parallel machines. PVM was developed by a collaboration between Oak Ridge National Laboratory, the University of Tennessee, and Emory University, all in the United States of America. PVM is public domain software.

PVM consists of two parts. A deamon process 'pvmd' runs on all nodes of the virtual machine. The PVM deamon handles communication between nodes, provides a robust fault tolerance to the virtual machine, and creates new PVM processes. The second part of PVM is a library, containing routines to be called by the user program. The routines are used to send and receive messages, to spawn processes or to modify the virtual machine. PVM3, which we have used, implements the SPMD style of parallel programming.

Point-to-point data transfer is performed by send/receive pairs. The sending routine is asynchronous, the receiving routine may be synchronous or asynchronous. Before sending, the data must be translated to a machine-independent format and put into a buffer. The sender process sends the buffer to the deamon, which in its turn sends the buffer to the deamon running on the destination computer. Finally, the receiving deamon delivers the buffer to the receiving process. After receiving, the data is unpacked, and translated to machine-specific format.

The data transfer as described above is the most general form of data transfer in PVM, which is needed if one defines virtual machines consisting of heterogeneous architectures, loosely coupled by means of local- or wide-area networks. In many cases, however, it is possible to simplify the point-to-point data transfer considerably. Firstly, if the virtual machine contains processors of one type, it is not necessary to translate the data to a machine-independent format. Secondly, the user can force PVM to set up a direct link between the sending and receiving process, thus circumventing the deamons. This will result in a substantial increase in the performance of the point-to-point data transfer. The multicast routine, which sends data from one node to a set of other nodes, is always handled by the deamon.

The PVM design assumes that the nodes are connected by unreliable and unsequenced point-to-point data transfer facilities[22]. Therefore, to guarantee reliability and sequencing of data transfer, PVM communication is based on the UDP/IP protocol.

2.6. PVM on the Parsytec systems

PVM was recently ported to the Parsytec systems in a joint effort by Parsytec GmbH, Genias GmbH and our group[20]. The experiments on the GCel, as described in this paper, were carried out with a very early β-version of PVM for Parsytec systems. The PowerXplorer experiments, on the other hand, are carried out with the official release of PowerPVM (version 1.1), which is PVM for the Parsytec PowerPC-based systems.

Following the philosophy of PVM, the GCel is considered as one node of a virtual machine, with the deamon running on the front-end of the GCel. The PVM processes running on the GCel node are actually executed on the transputers. If a GCel process wishes to communicate with a process running on another node on the virtual machine, it will always be handled by the deamon. If point-to-point communication is performed between processes inside the GCel (i.e. communication between transputers), it will not go through the deamon, and the data translation is not carried out.

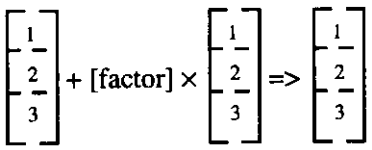
The multicast operation is implemented slightly differently than in standard PVM. If the processes participating in the multicast are all inside the GCel, the multicast will not invoke the deamon. This results in a major increase of performance of the multicast. This implementation of PVM on the GCel is coined 'heterogeneous PVM'.

PowerPVM for the PowerXplorer system is a so-called 'homogeneous PVM'. Here, the processors of the PowerXplorer are assumed to be the nodes of the virtual machine. Furthermore, the virtual machine can only consist of nodes in the PowerXplorer. The UDP protocol layer is omitted, and data translation before communication is not necessary. Furthermore, the deamons are reduced to very small processes to startup the virtual machine inside the PowerXplorer. As a user, the functionality of this PowerPVM, and the commands needed to start a virtual machine, are equal to the heterogeneous PVM. However, communication latencies are reduced significantly.

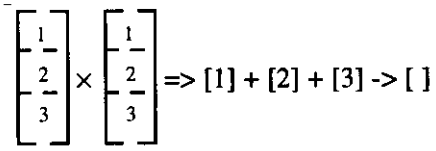
3. THE COUPLED DIPOLE APPLICATION

3.1. Functional aspects

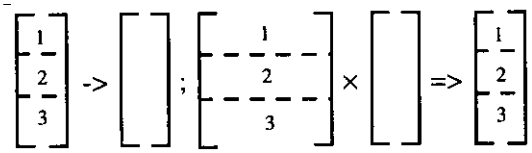
The coupled dipole (CD) method[23] simulates elastic light scattering from arbitrary particles. The particle is discretised in N small subvolumes, called dipoles. The simulation



3.a: parallel vector update



3.b: parallel inner product.



3.c: parallel matrix vector for a rowblock decomposed matrix.

Figure 3. A schematic drawing of the parallel implementation of the numerical operations. The decomposition of the vector and matrix is symbolised by the dashed lines; a single arrow (->) means a communication, and the implication mark (=>) means a (parallel) calculation

consists of two stages. Firstly, the internal electric field on the dipoles is calculated, and secondly the scattered electric field is calculated using the previously obtained internal field. The first stage can be formulated as a matrix equation $\mathbf{Ax} = \mathbf{b}$, with $\mathbf{A}$ a dense, complex symmetric $3N \times 3N$ interaction matrix, $\mathbf{b}$ a known vector of length $3N$, and $\mathbf{x}$ the wanted vector of length $3N$ containing the internal electric field. To solve this matrix equation is the most demanding part of a CD simulation. The equation is solved by means of the CGNR method[24], which is a conjugate gradient method suited for complex non-Hermitian matrices.

The scattered field is calculated by summation of the radiation of all N dipoles at the observation point. For every dipole one must calculate a matrix vector product $\mathbf{Bx}_i$, with $\mathbf{B}$ a 3×3 complex matrix and $\mathbf{x}_i$ a subvector, from the large vector $\mathbf{x}$, with length 3. The main computational difficulty of the CD method lies in the fact that for realistic simulations the number of dipoles N has to be very large[15]; typical values for N are 10^4–10^5. The calculation of the internal field is the most demanding part of the CD simulation.

We have developed a parallel version of the CD method[15]. Parallelism was introduced by means of a data decomposition: each processor receives N/p dipoles (p is the number of processors). In conjunction with the CGNR method, this decomposition is equivalent with a row-block decomposition of the interaction matrix $\mathbf{A}$[15]. We have studied parallel versions of the CGNR method in great detail[25], and concluded that parallel CGNR with a rowblock decomposition is very efficient if the number of rows (i.e. the number of dipoles) per processor is large enough.

The CGNR method is an iterative method to solve matrix equations. One iteration contains two matrix–vector products, three vector updates ($\mathbf{y} = a\mathbf{x} + \mathbf{y}$) and three inner products ($r = \mathbf{x} \bullet \mathbf{x}$). Figure 3 shows schematically how these operations are performed in parallel. The vectors represent electric fields on the dipoles, and are distributed among the processors. Every processor has subvectors of length $3N/p$.

The vector update can be performed completely in parallel; all data are present in the local memory of the processors. The inner product is calculated in two steps. First, all processors

calculate a partial inner product from their local vector data. This partial inner product is sent to all other processors and all the results are summed (a scalar accumulate operation). The rowblock decomposition of the matrix dictates how the parallel matrix-vector product is executed. First the argument vector, which is distributed among all processors, must be completely known by all processors. This means that all processors must send their part of the argument vector to all other processors. We will refer to this as a vector gather operation. After this vector gather the matrix–vector product can be performed in parallel. The resultant vector is correctly distributed among the processors.

Assume that one floating-point operation takes τ_{calc} μs. The execution time for the parallel vector update is (remember, all elements are complex numbers)

$$T^{vu}(N,p) = 24\lceil\frac{N}{p}\rceil\tau_{calc} \tag{1}$$

where $\lceil x\rceil$ is the ceiling function of x. The parallel inner product has an execution time

$$T^{ip}(N,p) = \left(24\lceil\frac{N}{p}\rceil - 2\right)\tau_{calc} + t_{sa} \tag{2}$$

where t_{sa} is the time to perform the scalar accumulate operation. Finally, the execution time of the parallel matrix–vector product with row block decomposition of the matrix is

$$T^{mv}(N,p) = 3\lceil\frac{N}{p}\rceil(24N - 2)\tau_{calc} + t_{vg} \tag{3}$$

where t_{vg} is the time for the vector gather operation. In real CD simulations the decomposed matrix **A** cannot be kept in the local memory of the processors. Fortunately we can compute the matrix elements as they are needed. This is exactly what happens in the experiments described in this paper. Therefore, an additional time to compute the matrix elements has to be included in Equation (3). Every 3×3 block in **A** describing an interaction between two dipoles requires 107 floating-point operations to be calculated. This results in

$$T^{mv}(N,p) = 3\lceil\frac{N}{p}\rceil(24N - 2)\tau_{calc} + 107N\lceil\frac{N}{p}\rceil\tau_{calc} + t_{vg} \tag{4}$$

Assume that the scattered field is calculated at w observation points (for instance scattered field as a function of scattering angle). This means that every processor has to calculate the contribution of its local dipoles to the scattered field at w observation points. To find the total scattered field, the fields calculated in all processors must be accumulated and summed. The total execution time for the parallel calculation of the scattered fields is therefore

$$T^{sf}(N,p) = w\lceil\frac{N}{p}\rceil(66 + 107)\tau_{calc} + t_{va} \tag{5}$$

The factor 66 is due to the complex matrix–vector product $\mathbf{Bx}_i$, and the factor 107 is due to the calculation of the matrix elements of **B**.

The time for one iteration of the CGNR method is

$$T^{iter} = 2T^{mv} + 3T^{vu} + 3T^{ip} \tag{6}$$

Assume that the CGNR method requires v iterations to find the solution vector **x**. The total execution time for one coupled dipole simulation T^{CD} is

$$T^{CD} = vT^{iter} + T^{sf} + T^{IO} \tag{7}$$

Here we included the time T^{IO} needed for input–output operations.

We have now specified the execution time of the parallel coupled dipole simulation in terms of floating-point operations and global communications. Both the global communications and the cost of a floating-point operation depend on the specific programming environment. If we can measure τ_{calc} and the global communication cost we can estimate the performance and scalability of the parallel simulation.

3.2. Implementation

3.2.1. General

Starting with the sequential coupled dipole C-source code, the parallel implementation in Parix, Express and PVM is straightforward. First, some bookkeeping code to establish the data decomposition must be implemented. Subsequently, the loop indices in the matrix–vector products, inner products and vector updates must be adapted to match the data decomposition. These additions to the source code are independent of the programming environment, except for one call to obtain the number of processors available in the network, and to find the location of each processor in the network.

Next, the communication routines must be included in the code. Before executing a parallel matrix–vector product the argument vector must be gathered in each processor. After calculating partial inner products in parallel, a scalar accumulate operation must be performed. Finally, after calculating the scattered field of the local dipoles in parallel, the resulting field must be accumulated. The implementation of these routines strongly depends on the programming environment, and will be discussed in the following Sections.

The parallel coupled dipole method was first implemented on a Meiko computing surface, containing 64 T800 transputers[14,25]. This implementation was in Occam, using the Occam Programming System. This code was ported to the Iserver environment to run on the Parsytec GCel. The structure of the implementation in Parix (in the C language) strongly resembles the Occam version. The Parix version was ported to Express, and PVM by adapting the communication routines and part of the bookkeeping code. We first discuss the Occam version, followed by the Parix, Express and PVM versions of the program.

3.2.2. The Iserver–Occam program

The coupled dipole method with the rowblock decomposition of the interaction matrix was implemented on a bidirectional ring topology. The vector gather operation is implemented as follows:

1. In the first step every processor sends its local part of the vector to the left and the right processor and, at the same time, receives the local part from the left and the right processor.

2. In the following steps, the parts received in a previous step are passed on from left to right and vice versa, and in parallel, parts coming from left and right are received and stored.

After $\lfloor p/2 \rfloor$ steps (with $\lfloor x \rfloor$ the floor function of x), every processor in the ring has received the total vector. The scalar and vector accumulates are implemented likewise, but in addition the processor must summate the received data with its local data.

The pseudo-Occam code for the vector gather operation is given below:

```
Vector Gather
PAR
  SEQ -- from right to left
    PAR -- start the communication
      send to left
      receive from right
    PAR i=1 FOR "p/2-1" -- pass data from right to left
      send to left which was received from right in the previous step
      receive from right
  SEQ -- from left to right
    PAR -- start the communication
      send to right
      receive from left
    PAR i=1 TO "p/2-1" -- pass data from left to right
      send to right which was received from left in the previous step
      receive from left
```

The operation consists of two parallel branches, one receiving data from the right and sending data to the left, the other vice versa. Each branch consists of a communication initialisation, which is the first PAR inside the SEQ branches. Subsequently the gather operation is carried out in the replicated PAR operation. The Occam language is very powerful to express such complicated communication patterns. Owing to the close relationship between Occam and the transputer, this part of the code runs very efficiently on a transputer (see Section 4). The complete Occam source code of the vector gather operation is, however, much more complicated due to the complex datastructure needed to store the vector with complex variables and due to the small load imbalance which is introduced if the number of dipoles N is not a multiple of the number of processors p.

In our application, all transputers run two processes, a router and a calculator. Router processes on neighbouring transputers are connected by a channel. These channels are associated with hardware transputer links via a configuration file. The router process calls communication routines, such as the vector gather operation. The calculator process performs the work on the decomposed data. This work is divided in cycles, at the end of every cycle a communication step occurs. The calculator sends a command, in the form of a single character, to the router process. The router process receives this character, interprets it and issues the desired communication routine. During this communication step the calculator process is idle. After finishing the communication, the router process sends a 'ready' signal to the calculator process, which then proceeds with the next cycle in the algorithm.

The original implementation of the Occam code for the Meiko system was carried out in approximately eight months. Porting of the code to the Iserver environment took approximately three weeks. The total Iserver–Occam source code contains 4500 lines.

3.2.3. The Parix program

The structure of the Parix implementation, developed in C, is similar to the Occam program. A virtual ring topology is defined by a call to the virtual topology library, and two threads

are started on each transputer, a calculator and a router thread. The main program is shown globally below:

```
main ()
{
     /* main of parallel CD method */
     MakeRing (...)        /* Create Ring Topology */
     GetRing_Data (...);  /* Extract information of Ring topology */
     StartThread (Calculator,...);  /* Start the Calculator thread */
     StartThread (Router,...);           /* Start the Router thread */
     Wait for threads to terminate
}
```

First, the ring topology is established; next the location in the ring is found by the GetRing_Data call, and the two threads are started.

We could have mimicked the vector gather operation as described in the previous Section using Parix's asynchronous communication calls. However, to start an asynchronous communication in Parix is very expensive, and would result in prohibitive startup costs. Therefore we redesigned the vector gather operation. First, the data are sent in just one direction, from left to right, and the send and receive operations are synchronous. To achieve this, the vector gather operation must start two threads: one to receive data from the left and a second to send the data to the right. Furthermore, a mechanism to synchronise the receiving and sending thread must be implemented.

The final vector gather operation is shown below:

```
Vector Gather
{
    StartThread (Send_to_Right,...);     /* Start sending thread */
    StartThread (Recv_from_Left,...);    /* Start receiving thread */
    Wait for threads to terminate
}

Send_to_Right
{
     /* pass data from left to right */
     for (i=0 to p-1) {
            Send (to right);
            synchronise with receiving thread
        }
}

Recv_from_Left
{
        /* pass data from left to right */
        for (i=0 to p-1) {
               Recv (from left);
               synchronise with sending thread
        }
}
```

The synchronisation is implemented by establishing a virtual link between the sending and receiving thread, and communicating *ready* signals between both threads.

This parallel program was developed using the existing sequential C code, and the experience we gained during the development of the Occam code. The parallel program was developed in two weeks, the final version of the communication routines as described above consumed two more weeks, and optimisation of the numerical parts of the application, especially the matrix–vector product, lasted another three weeks. The complete C source code contains 3800 lines. The Parix program runs without modifications on the GCel and the PowerXplorer system.

3.2.4. The Express program

The Express program is based on the optimised Parix C program. The main structure of the Express program, however, is very different from the Parix implementation. Express offers global communication routines which are used in our implementation. These global communications free the application programmer from the notion of a topology. The Express program consists of p parallel processes communicating via (global) message-passing routines. The Express kernel takes care of placing the processes on processors and of the actual routing of messages through the network.

The Express program consists solely of the calculator process. The calls in the original Parix calculator to the router are replaced by calls to global communication routines of the Inter Process Communication library of Express. Of course we could have implemented the communication routines using constructs like the Parix program, but we feel that a typical application programmer *will* use the global communication routines if they are available. The numerical part of the code remained unchanged.

The vector gather operation is implemented using the *exconcat* function. This function concatenates the data of each participating node into a single buffer. If the data of each participating node are not equally sized (that is, if N/p is not an integer) the result of the *exconcat* will be a buffer with 'empty' places in it. Therfore, the resulting buffer must be compacted. The vector gather operation is now reduced to

```
Vector Gather
{
     exconcat ();
     if necessary, compact the resulting buffer
}
```

The accumulate operations are implemented using the *excombine* function, which allows each node to combine data from all other nodes using a user-supplied function (i.e. in this case addition).

Express uses buffers for the communication routines. The number of buffers and their size can be adjusted by the user, and large buffers result in better communication characteristics. However, it turned out that we had to choose very small buffers of 1 Kbyte to be able to run Express on large partitions. This resulted in a small performance degradation of the communication routines.

The time to port the Parix program to Express was three days. The resulting Express version of the coupled dipole code runs without change on a cluster of workstations, on the Meiko Computing Surface and the Parsytec GCel. The total source code contains 2400 lines.

3.2.5. The PVM program

The PVM program is also based on the optimised Parix C program, and has the same main structure as the Express implementation. PVM offers a global communication routine (the *multicast*) which is used to implement the vector gather operation. The PVM program only consists of the calculator process, with the calls in the original Parix calculator to the router replaced by calls to global communication routines. The numerical part of the code remains unchanged.

The vector gather operation is implemented using the *multicast* function. This function sends data from the calling node to all participating nodes. Therefore, the vector gather operation can be implemented by issuing a *multicast* on all processors, followed by p-1 receive operations to obtain the data from all other processors:

```
Vector Gather
{
   prepare data for sending
   pvm_mcast (...) /* the multicast operation */
   for(i=0; i<p; ++i) {
           pvm_recv(...)    /* receive data from other nodes */
           unpack data and put into buffer
   }
   if necessary, compact the resulting buffer
}
```

Before sending, the data are translated to a machine-independent format (the preparation stage), and after all receives the data have to be translated into the transputer format. Finally, as with the Express implementation, the resulting buffer is compacted if necessary.

The time to port the program to PVM was eight days. The resulting PVM version of the coupled dipole code runs without change on a cluster of workstations, the Parsytec GCel, and the Parsytec PowerXplorer. The total source code contains 2500 lines.

4. RESULTS

4.1. Methodology

The run time of the parallel coupled dipole implementation is determined by τ_{calc} and the global communication times (see Section 3.1). In the Iserver–Occam and Parix implementation the global routines are explicitly implemented using (nearest-neighbour) point-to-point communications. The global routines in Express and PVM are also implemented using basic point-to-point communication, since the hardware does not support collective communications. However, as an application programmer we do not exactly know how these global routines are actually implemented.

To compare the environments we have measured floating-point performance (i.e. τ_{calc}), the basic communication routines, the global communication routines and finally the execution time of the parallel coupled dipole method on the Parsytec GCel. Subsequently we have compared PowerPVM and Parix on the PowerXplorer system by measuring floating-point performance and relevant communication routines.

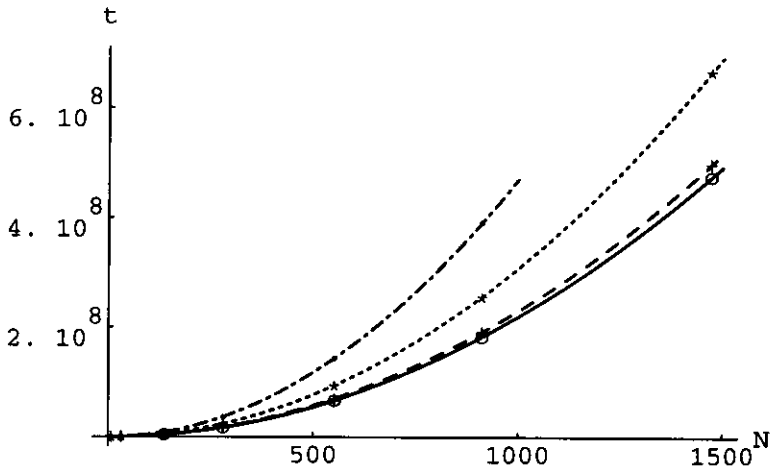


*Figure 4. The execution time (in μs) of the matrix–vector product on one processor, as a function of the number of dipoles. The ticks are the measurements: o for PVM, * for Express, + for Parix and • for Iserver. The lines are the fits: PVM (—), Express (· · ·), Parix (– – –) and Iserver (- · - · -)*

4.2. Floating-point performance on the Parsytec GCel

We have modelled the floating-point performance with just one parameter τ_{calc} (Section 3.1). Beside the raw power of the floating-point unit of the T805, this parameter should also include the effects of loop overhead, memory access, cache behaviour, etc. Clearly τ_{calc} depends on the type of operation that is performed, and one can argue that this single-parameter model is to simple to predict the floating-point performance. However, by keeping our specific application in mind we can use this single parameter.

The numerical work in the coupled dipole method consists of matrix–vector products, inner products and vector updates. Since the largest portion of the numerical work is the matrix–vector product, we have measured the execution time of the matrix–vector product on one processor, as a function of the number of dipoles N. The parameter τ_{calc} was extracted from the measurements by fitting them with equation (4).

Figure 4 shows the results of the measurements on the GCel, together with the fitted functions. The τ_{calc} that resulted from the fits are: for Parix τ_{calc} = 1.28 μs/flop, for Iserver–Occam τ_{calc} = 2.63 μs/flop, for Express τ_{calc} = 1.72 μs/flop, and for PVM τ_{calc} = 1.25 μs/flop.

4.3. Basic point-to-point communication on the Parsytec GCel

We assume that point-to-point communication can be described by a linear two-parameter model. The point-to-point communication time t_{pp} is

$$t_{pp} = \tau_{setup} + n\tau_{send} \tag{8}$$

with n the number of bytes sent, τ_{setup} a setup time to initialise and start the communication, and τ_{send} the send time to transfer 1 byte. Here we have neglected effects due to buffering.

In Occam all communication is synchronous and between neighbouring transputers. We have measured t_{pp} as a function of n. The results were perfectly linear (data not shown), and were fitted to equation (8) with a least-squares method. We could distinguish three different situations: communication between transputers inside one atom (see Figure 2), communication between neighbouring transputers in two adjacent atoms inside one cube, and communication between neighbouring transputers in two adjacent cubes. Table 1 shows the results of the fits, together with estimates of the error. The setup time is almost constant; the send time increases with increasing 'distance' between the nodes.

Table 1. Send and setup times for nearest-neighbour point-to-point communication for Iserver–Occam

Communication	τ_{send} (μs/byte)	τ_{setup} (μs)
inside atom	0.71 ± 0.01	3.73 ± 0.02
between atoms inside cube	0.87 ± 0.01	3.82 ± 0.02
between cubes	0.90 ± 0.01	3.84 ± 0.02

Parix's virtual links allow point-to-point communication between any node; the kernel of Parix routes the messages through the hardware. However, in the coupled dipole implementation (see Section 3.2) the only point-to-point communication consists of synchronous send/receive pairs between adjacent processors in the virtual ring topology. Therefore we have only measured synchronous point-to-point communication between neighbouring processors in a virtual ring topology. In most cases the virtual ring can be mapped onto the physical two-dimensional mesh such that neighbouring processors in the virtual ring are also neighbouring processors in the physical two-dimensional mesh.

We have analysed the same nearest-neighbour communications as with Iserver–Occam. However, we could not distinguish significant differences between the results; in all cases we find $\tau_{send} = 0.92 \pm 0.02$ μs/byte and $\tau_{setup} = 67 \pm 2$ μs.

The analysis of point-to-point communication in Express and PVM is complicated by two effects. Firstly, the localisation of the parallel processes is not known, and therefore we do not know if a communication is between physical neighbouring processors. Secondly, in the Express and PVM program we only use global communication primitives. Although these global communication functions are implemented using point-to-point communication, we do not know in detail *how* these global functions operate. To obtain a picture of the basic communication performance of Express and PVM we measured all possible point-to-point communications between node zero and all other nodes in an Express and PVM partition, and analysed the distribution in the resulting setup and send times.

Figure 5 shows all measured point-to-point communications in a 256-node Express and PVM partition. We have fitted all experiments and generated histograms of τ_{send} and τ_{setup}. The histograms are drawn in Figures 6 and 7.

For both PVM and Express we see two fast connections having a τ_{send} of approximately 0.9 μs/byte and 1.0 μs/byte. These numbers are comparable to the τ_{send} of Parix. The rest of the connections of Express clusters around 1.53 μs/byte, 1.73 μs/byte and 1.88

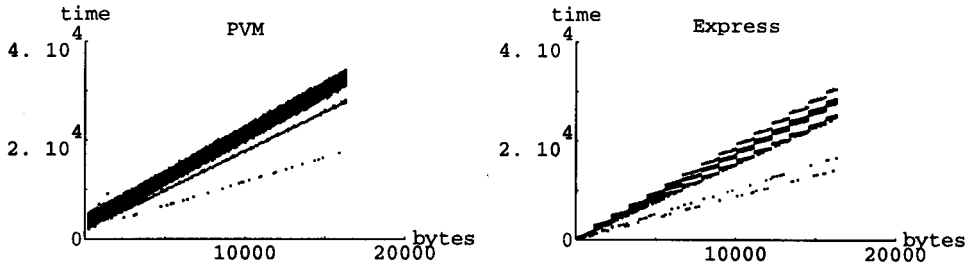


Figure 5. All measured point-to-point communication times (in μs) as a function of the number of transferred bytes for a 256-node partition in the GCel

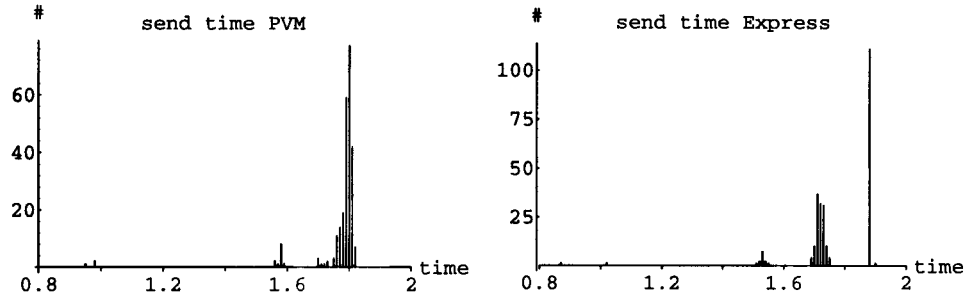


Figure 6. The histogram of the occurrences of send times (in μs/byte) in a 256-node partition in the GCel; the step size is 0.01 μs/byte

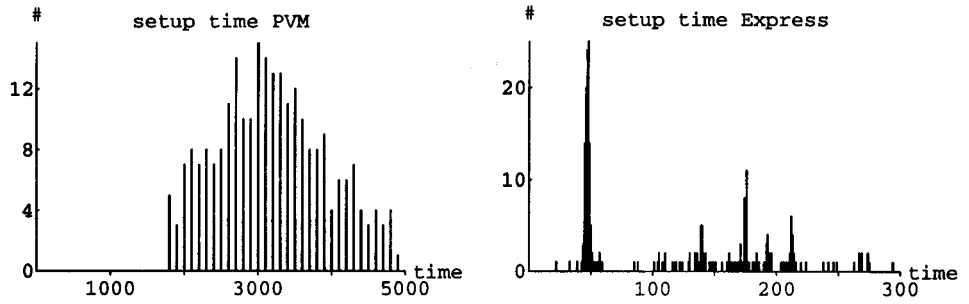


Figure 7. The histogram of the occurrences of setup times (in μs) in a 256-node partition in the GCel; the step size is 1 ms for Express and 100 μs for PVM

μs/byte. Most of the PVM connections cluster around 1.8 μs/byte. In conclusion, the τ_{send} of PVM and Express are comparable. Clustering of τ_{setup} of Express is less obvious; a large peak around τ_{setup} = 48 μs can be seen, and a very broad distribution between 100 and 300 μs. PVM shows a broad distribution around ≈3000 μs. Here we clearly observe a difference with the other environments. PVM has much higher setup times for point-to-point communication than the other three environments.

4.4. Global communication on the GCel

We have measured the time for the vector gather operation as a function of the number of processors and as a function of the total vector length n in bytes. For Iserver–Occam and Parix we know exactly how the vector gather operation is implemented, and we can formulate a model for the execution time of the vector gather. In Iserver–Occam we have to perform $\lfloor p/2 \rfloor$ point-to-point communications between neighbours on the ring. In each communication step $\lceil n/p \rceil$, bytes must be transferred. Therefore,

$$(t_{vg})^{Iserver} = \lfloor p/2 \rfloor (\tau_{setup} + \lceil n/p \rceil \tau_{send}) + \tau_{init} \tag{9}$$

In equation (9) we have included a term τ_{init} which describes the initialisation time for the vector gather operation. The setup and send times are known from the point-to-point measurements.

In Parix the vector gather operation is almost identical to the Iserver version, but now the data are transferred through a monodirectional ring and therefore we have to carry out $(p-1)$ point-to-point communications. This results in

$$(t_{vg})^{Parix} = (p-1)(\tau_{setup} + \lceil n/p \rceil \tau_{send}) + \tau_{init} \tag{10}$$

Although Express' exconcat function is described for hypercube architectures by Fox *et al.*[19] [chapter 14], we will not attempt to find a detailed model for the exconcat as implemented on the two-dimensional grid of the Parsytec GCel. We assume that the exconcat is linear in n because is uses point-to-point communications which are linear in n. The same is assumed for the vector gather operation as implemented with PVM's multicast function.

For each value of p we measured communication time as a function of n. We have fitted the measurements to $t_{vg} = \tau_a n + \tau_b$; Table 2 shows the results of the fit.

For the Iserver and Parix results, the goodness of fit, as defined by, for example, Press *et al.*[26] [chapter 14] was in all cases very close to 1, indicating that the linear model is satisfactory to describe the global communication routines. The errors in τ_a and τ_b for the Iserver and Parix results are estimated to be smaller than 1%.

Table 2. Result of fitting the global communication time as measured on the GCel to a linear model $t_{vg} = \tau_a n + \tau_b$

p	τ_a (μs/byte)				τ_b (μs)			
	Iserver	Parix	Express	PVM	Iserver	Parix	Express	PVM
2	—	1.0	0.6	3.2	—	9.8×10^2	2.8×10^2	6.6×10^2
4	0.3	0.9	1.8	3.6	5.0×10^2	1.2×10^3	−92	2.7×10^3
8	0.4	0.9	4.2	3.5	5.3×10^2	1.2×10^3	8.3×10^2	2.1×10^4
16	0.5	0.9	5.6	3.8	6.9×10^2	3.4×10^3	1.5×10^4	4.5×10^4
32	0.5	1.1	13	5.0	1.3×10^3	6.1×10^3	8.4×10^4	9.5×10^4
64	0.5	1.0	20	9.3	2.8×10^3	1.2×10^4	3.1×10^5	1.7×10^5
128	0.5	1.1	32	17	6.1×10^3	2.3×10^4	1.3×10^6	5.6×10^5
256	0.5	1.1	60	21	1.3×10^4	5.3×10^4	6.0×10^6	1.9×10^6
512	0.5	1.1	—	—	2.6×10^4	1.0×10^5	—	—

The situation for Express and PVM is quite different. The Express measurements show a very clear buffering effect, resulting in jumps in the execution time if the buffers are completely filled. The reported fit results are based on measurements before the first jump occurs. The PVM measurements show a linear relationship as long as $n > 1000$ bytes. For smaller n we observe a levelling-off of the data. In this case the fits are based on measurements for $n > 1000$ bytes.

Comparison of the τ_a and τ_b for Iserver–Occam and Parix with equations (9) and (10) shows that the observed behaviour is quantitatively in agreement with the models for the vector gather operation. The parameter τ_b is in both cases linearly dependent on p. However, fitting τ_b as a function of p to a line results in much larger values for τ_{setup}, for example, as those expected from the point-to-point communication measurements (data not shown).

The fitted values for τ_a of Iserver–Occam agree very well with the model, as can be seen by comparing Table 2 with Table 1 and equation (9). The agreement in the case of Parix is also satisfying, especially for large p.

We can observe two fundamental differences between the PVM and Express results with the Iserver–Occam and Parix results. First, τ_a depends on p, and increases sublinearly (probably as $\log(p)$ for Express) in p. Furthermore, the initialisation τ_b increases faster than linearly with p, which results, for large p, in (unacceptably) high initialisation times of 6 s, for example, for Express on a 256 partition. This result is a very good example of the tradeoff of programmability against performance. The Express and PVM vector gather operations consist of just one call to a global communication routine, and trivial buffer compacting. However, the price to be paid is bad scaling behaviour, as compared to Iserver–Occam or Parix.

4.5. Performance of the coupled dipole implementation on the Parsytec GCel

We have measured the execution time for one conjugate gradient iteration, for the calculation of the scattered field and for a total coupled dipole simulation (including I/O) as a function of the number of dipoles N in the simulation and the number of processes. The number of dipoles was limited to 2176 to prevent prohibitive execution times on one processor.

We only show the result for the execution time of one conjugate gradient iteration (see figure 8). The time for one iteration is shown for $N = 32$, $N = 552$ and $N = 2176$. All other problem sizes show comparable behaviour. Note that in all following figures the solid lines are for PVM, the short-dashed lines for Express, the long-dashed lines for Parix, and the dotted-dashed line for Iserver. Furthermore, note that in Figure 8 both the number of processors and the execution time are drawn on a logarithmic scale.

From the measured execution times one can calculate speedups and efficiencies. We show the efficiencies of one conjugate gradient ($N = 32$ and $N = 2176$, Figure 9) and of the total coupled dipole simulation ($N = 32$ and $N = 912$, Figure 10).The efficiency is defined as $T(p = 1)/(pT(p))$, with $T(p)$ the execution time on p processors.

4.6. Comparison of Parix and PowerPVM on the Parsytec PowerXplorer

From the experiments in Sections 4.1–4.5 we can conclude that a measurement of the environment-dependent building blocks results in a good first comparison between different parallel programming environments (see also Section 5, discussion). Therefore, in this Section we present measurements of the τ_{calc} and the point-to-point and global communca-

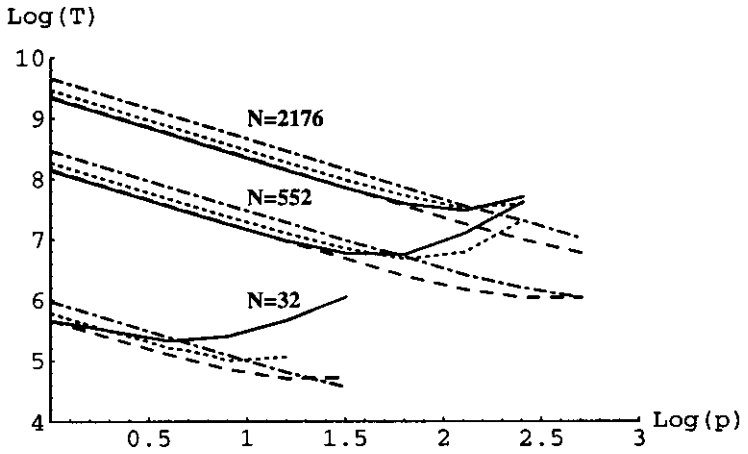


Figure 8. The execution time of one iteration of the conjugate gradient method, as a function of the number of processors, for N = 32, N = 552 and N = 2176: PVM (—), Express (· · ·), Parix (– – –) and Iserver (- · - · -)

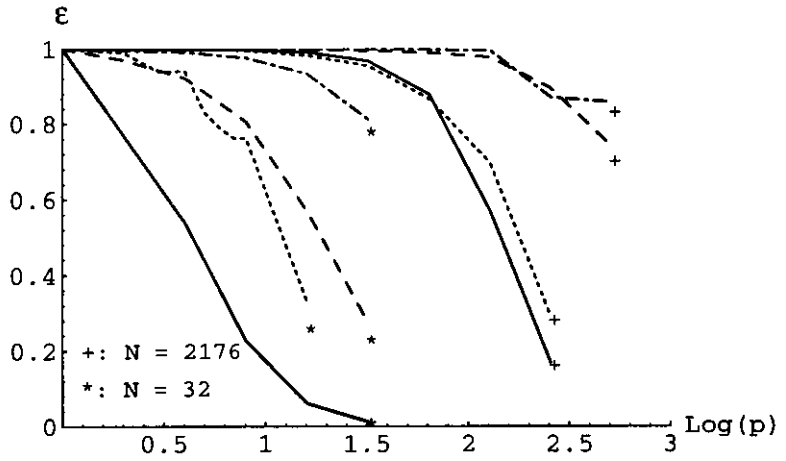


Figure 9. The efficiency of one conjugate gradient iteration, as a function of the number of processors, for N = 32 and N = 2176: PVM (—), Express (· · ·), Parix (– – –) and Iserver (- · - · -)

tion times of Parix and PowerPVM on the Parsytec PowerXplorer. These measurements will give us a good insight of the performance of the large class of applications such as the coupled dipole method, which are characterised by very regular data structures and kernel routines such as a parallel matrix–vector product.

We have measured a τ_{calc} of 0.08 μs for both Parix and PowerPVM, resulting in a performance of 12.5 Mflop/s per PowerPC-601. For small vectors, fitting completely in the 32 Kbyte cache of the PowerPC, we have measured a performance of 54 Mflop/s. However,

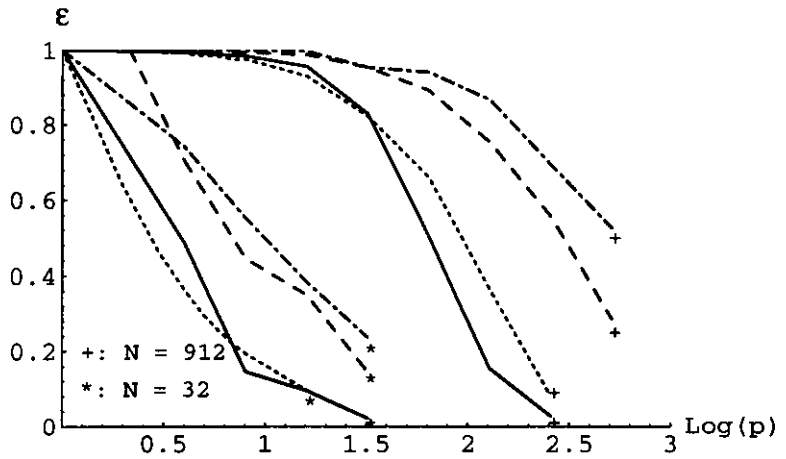


Figure 10. The efficiency of a total coupled dipole simulation (including I/O) as a function of the number of processors, for N = 32 and N = 912; PVM (—), Express (· · ·), Parix (– – –) Iserver (- · - · -)

in normal production runs the vectors will usually not fit in the cache.

Figures 11 and 12 show the histograms for τ_{send} and τ_{setup} (as introduced in Section 4.3), in the 32-node partion of the PowerXplorer system for Parix and PowerPVM.

Table 3 shows the results of fitting the communication time of the vector gather operation in Parix and PowerPVM to $t_{vg} = \tau_a n + \tau_b$.

Table 3. Result of fitting the global communication time as measured on the PowerXplorer to a linear model $t_{vg} = \tau_a n + \tau_b$

p	τ_a (μs/byte)		τ_b (μs)	
	Parix	PowerPVM	Parix	PowerPVM
2	0:6	0.7	—	7.7 102
4	0.7	1.2	7.2×10^2	9.4×10^2
8	0.9	1.6	1.3×10^3	9.1×10^3
16	1.0	2.2	1.9×10^3	8.6×10^3
32	1.0	3.7	5.1×10^3	1.3×10^4

5. DISCUSSION

This paper compares native and generic parallel programming environments, i.e. Iserver–Occam and Parix as against Express and PVM. An exhaustive comparison between the complete functionality of all four environments is not very useful. We need a guideline in the form of an application that must be, or is, implemented in these environments. Our particular application, the coupled dipole simulation, relies on a representative subset of the complete functionality of the environments. The coupled dipole method can be viewed as

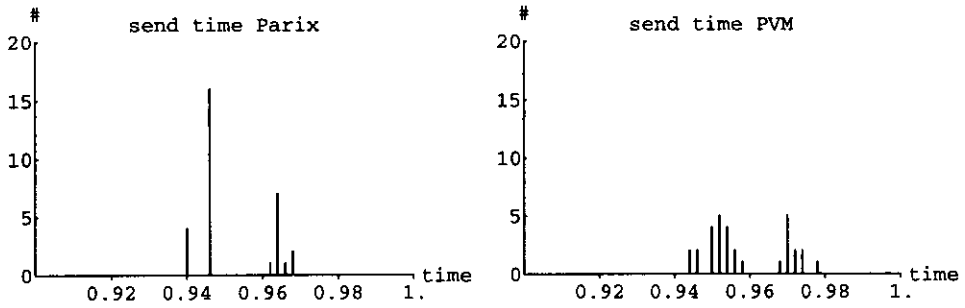


Figure 11. The histogram of the occurrences of send times (in μs/byte) in the 32-node partition in the PowerXplorer; the step size is 0.01 μs/byte

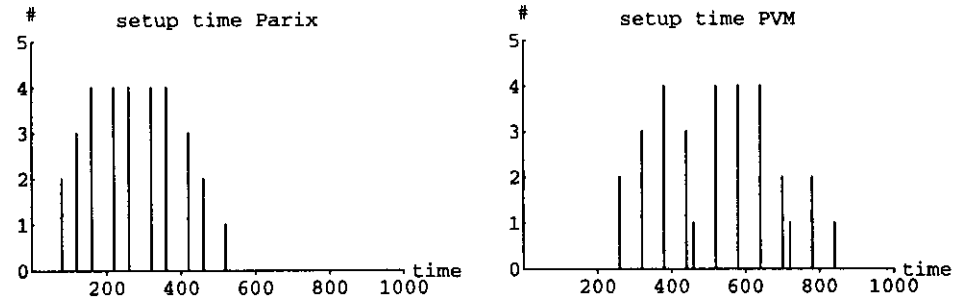


Figure 12. The histogram of the occurrences of setup times (in μs) in the 32-node partition in the PowerXplorer; the step size is 10 μs

a representative of a large class of applications relying on iterative methods to solve large systems of linear equations.

We analysed basic and global communication routines which are needed in the coupled dipole program, floating-point performance and the execution time of the application on the Parsytec GCel. In this discussion we compare the environments on the first three levels, and investigate if we can predict the performance of the coupled dipole program. As will be shown, this is indeed possible and this result shows that our basic measurements can be used as a heuristic to assess the merits of parallel programming environments. We have subsequently applied this heuristic to compare Parix and PowerPVM on a PowerPC platform.

The floating-point performance, which in our definition is the lumped sum of floating-point unit performance, loop overheads, memory access overheads and others, is basically a test of the compilers. We timed a computational kernel of the coupled dipole method, and therefore the resulting numbers should not be compared with other results for floating-point performance. We can, however, compare the results of the same computational kernel compiled by the different compilers. PVM and Parix have the best performance, with a τ_{calc} of 1.25–1.28 μs/flop. Express has the second best performance with τ_{calc} = 1.72 μs/flop and surprisingly the Iserver–Occam result is much worse with a τ_{calc} = 2.63 μs/flop. This large τ_{calc} for Iserver–Occam is probably due to the way in which we were forced to

implement the double-precision complex matrix–vector product, resulting in bad memory access behaviour.

The point-to-point communication results for the GCel show the power of Iserver–Occam. A setup time of only 3.8 μs, compared to 67 μs for Parix, 50 $\mu s < \tau_{setup} < 300$ for Express, and 2000 $\mu s < \tau_{setup} < 5000$ μs for PVM, shows that the Occam channels are directly mapped onto the transputer hardware. The abstractions of Parix and Express induce much larger setup times. The setup for PVM is extremely high, due to the presence to the UDP layer (see Sections 2.5 and 2.6). We should mention that newer versions of PVM on the Parsytec GCel, which use a heavily optimised intermediate communication kernel, will have much better communication performance. First measurements on this optimised heterogeneous PVM for the Parsytec GCel suggest drastically reduced latencies, comparable to PowerPVM for the PowerXplorer (data not shown; for more information see [20]).

The same is true for the sending time, but here the difference between Iserver–Occam and Parix is not that large, and the fastest connections in the Express and PVM partition also compare very well with Iserver–Occam and Parix. Still, the situation as drawn in Figure 1 is confirmed for point-to-point communication.

The global communication routine reveals the same picture: decreasing performance as the price to be paid for decreasing hardware visibility and increasing programmability. The Iserver–Occam implementation has the smallest initialisation time and fastest transfer rate. The initialisation time in Parix is a factor 2 to 4 larger than in Iserver–Occam, and the transfer rate is a factor 2 smaller.

Except for very small partitions the Express and PVM vector gather operation are always slower than Iserver–Occam and Parix. Furthermore, as shown in Section 4.4, the global communication in Express and PVM has a fundamentally different scaling behaviour, i.e. an increasing τ_a as a function of p, and a faster than linear increase of τ_b as a function of p. This results in transfer rates of 60 μs/byte[2] and initialisation times in the order of seconds! In compute-intensive applications like the coupled dipole simulation, where the computational work scales with N^2, this does not need a problem (as can be seen in the sequel). However, for applications where the computational work is linear in N, and which need such global communications, this scaling behaviour will result in very bad performance on massively parallel systems.

Figure 8 shows the execution time of one iteration as a function of p and N. The calculation time is of order $O(N^2 / p)$ (see Section 3.1), whereas the total communication time is of order $O(N)$. Therefore, for large N and small p we actually observe the floating-point performance of the environments. The Parix and PVM implementation are the fastest. Express is second, and due to its poor floating-point performance the Iserver–Occam implementation is the slowest. However, if p is increased for a constant N, the communication time becomes more important, and at some point becomes larger than the calculation time. Due to the very poor performance of the vector gather function of Express and PVM, the results already show an increase in execution time of one iteration for $p = 100$ and $N = 552$, for example, whereas the execution time of the Iserver–Occam and Parix implementation still decreases. For an even larger number of processors the Parix implementation also shows an increase in execution time and for $N = 552$ the Iserver–Occam implementation has a

[2] Note, however, how this transfer rate τ_a is defined in Section 4.4. It is not the same as the link speed of the point-to-point communication, but is defined as the rate at which the total vector is transferred through the network in the vector gather operation.

comparable execution time to that of the Parix implementation.

Scalability of the parallel implementations is much better observed by looking at the efficiency plots. From Figures 9 and 10 we can see that in all cases the Iserver–Occam has the best efficiency, followed by Parix and finally by Express and PVM. We have to be very careful with this comparison. Iserver–Occam spends much more time in the parallel calculations and will therefore have a better efficiency.

The efficiency plots, however, allow one important conclusion, which holds for the parallel coupled dipole implementation in all four environments. If the number of dipoles is very large compared to the number of processors ($N / p >> 1$), then the efficiency can be very close to 1. However, for Express and PVM N/p has to be much larger than Iserver–Occam, for example, to reach efficiencies close to 1. In that situation, which is frequently encountered in real production runs, the execution time is mainly determined by τ_{calc}, and the Parix and PVM implementations are the fastest.

We should note that in principle we can implement the vector gather operation in Express and PVM in the same way as in Parix. This would induce a large programming overhead, but we may expect from the histograms as reported in Figures 6 and 7 that the vector gather operation of at least Express will then be comparable to the Parix results. Finally, the next versions of Parix also contain global communication routines. This greatly increases the programmability of global communications in Parix, but experiments will have to establish if the same price as in Express and PVM has to be paid.

Knowledge of the floating-point performance and the global communication capabilities of the environments allowed us to interpret and understand the execution time and efficiency of the application. Let us now investigate if we can predict the execution time of the coupled dipole application. We only show results for the execution time of one conjugate gradient iteration. We use equations (1), (2), (4) and (6) to calculate T^{iter} as a function of p and N. The times for the vector gather operation, and for the vector and scalar accumulate operation, are calculated by using the results of Table 2, and finally τ_{calc} as reported in Section 4.2 is used. Figures 13 and 14 show the comparison between the measurements (the points) and theory of the execution time of one conjugate gradient iteration, as a function of the number of dipoles N, for $p = 32$ and $p = 256$, respectively.

In all cases we can accurately predict the execution time of the Parix and Iserver–Occam implementation. For small partitions (i.e. $p = 32$, see Figure 13) we can also accurately predict the execution time of the Express and PVM implementation. However, for larger partitions errors between theory and experiment as large as 30% are observed (see Figure 14). Furthermore we observe that the predictions overestimate the execution time of the Express implementation and underestimate those of the PVM implementation. These errors can be traced back to errors in the fit parameters of the global communication (especially the parameter τ_b).

We can predict the execution time of our specific application in the different environments using basic performance measurements in these environments. Although the predictions for the PVM and Express implementations are not very accurate, we believe that the basic performance measurements, combined with a time complexity model as presented in Section 3.1 and information on programmability and portability, allows us to compare the environments. Table 4 compares the four environments on the Parsytec GCel on a scale from good (++) to poor (−−). The environments are judged on programmability (Prog.), on source level portability between hardware platforms (Port.), on the availability of tools, such as a performance analyser, or a debugger. These first three characteristics are

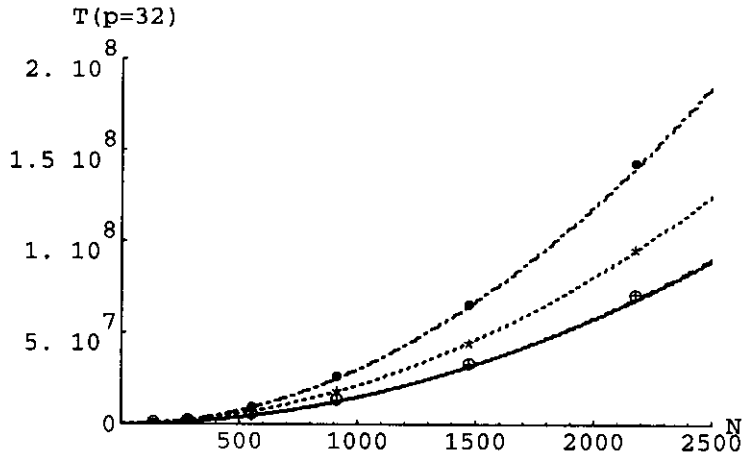


*Figure 13. The execution time (in μs) on 32 processors, for one conjugate gradient iteration, as a function of the number of dipoles N. The ticks are the measurements; o for PVM, * for Express, + for Parix and • for Iserver. The lines are the theoretical predictions: PVM (—), Express (· · ·), Parix (– – –) and Iserver (- · - · -)*

independent of the particular application. The rest of the columns are devoted to application-dependent characteristics. These characteristics are the floating-point performance (FP), the communication performance (Comm.), and the scalability, or efficiency, of the resulting parallel program (Scal.). Depending on the importance of portability as against scalability or floating-point performance, for example, an overall judgement of the environments can be made.

Table 4. A comparison of the Iserver–Occam, the Parix and the Express environment on programmability (Prog.), on source level portability (Port.), on the availability of tools (Tools), on floating-point performance (F.P.), on communication performance (Comm.), and on scalability, or efficiency, of the resulting parallel program (Scal.)

	Prog.	Port.	Tools	F.P.	Comm.	Scal.
Iserver	–	–	–	-	++	++
Parix	+–	-	-	++	+	+
PVM	+	++	+-	++	–	-
Express	++	++	+	+	–	-

Finally, in Section 4.6, we applied our heuristic to compare the generic PowerPVM against the native Parix on a new parallel system, the Parsytec PowerXplorer. The floating-point performance of one node in the PowerXplorer is 12.5 Mflop/s. The communication hardware, however, is still based on transputers. This means that in the particular architecture of the PowerXPlorer there is a slight imbalance in communication and computation, as compared to the GCel. It is therefore of utmost importance for generic environments,

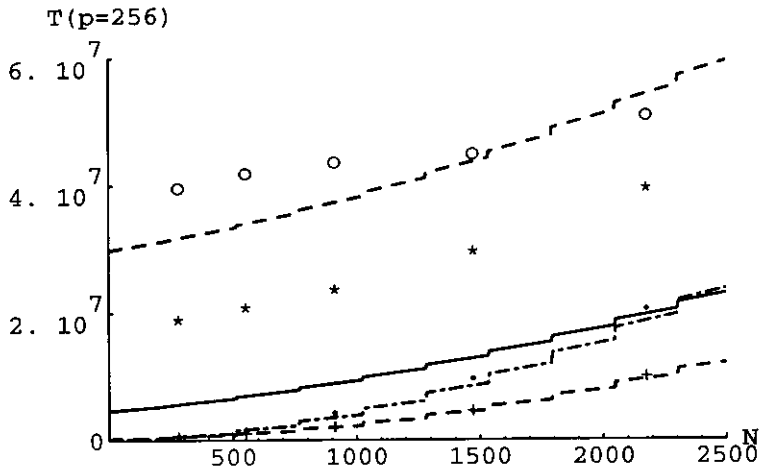


*Figure 14. The execution time (in μs) on 256 processors, for one conjugate gradient iteration, as a function of the number of dipoles N. The ticks are the measurements; o for PVM, * for Express, + for Parix and • for Iserver. The lines are the theoretical predictions: PVM (—), Express (· · ·), Parix (– – –) and Iserver (- · - · -)*

such as PowerPVM, not to introduce too much extra communication latency, since this would strengthen the imbalance between communication and computation. Our heurstic, based on measurements of the point-to-point histograms, as presented in Section 4.3, and global communication as presented in Section 4.4, is a valuable tool to compare generic PowerPVM with native Parix.

Figures 11 and 12 compare send and setup times for Parix and PowerPVM for a 32-node Parsytec PowerXplorer system. Setup times for PowerPVM are ≈1.5 times higher than those of Parix, whereas the send times are comparable. This means that point-to-point communication of PowerPVM is as good as in Parix; the extra PVM layer introduces no additional communication latencies.

The global communication of PowerPVM, however, is still much more expensive than the handcoded Parix vector gather operation (see Table 3). To our surprise, the vector gather operation in PowerPVM is almost as expensive as the vector gather operation in the heterogeneous PVM for the GCel (see Table 2), although the underlying point-to-point communication of PowerPVM is much faster (compare Figures 11 and 12 with Figures 6 and 7). This indicates that the implementation of the global operations is far from optimal.

Global communication routines strongly increase the programmability of parallel computers. However, in our case studies we have observed the poor scaling behaviour of these routines, as compared to handcoded Parix routines that exploit data locality and point-to-point communication on virtual topologies.

Currently the message-passing interface (MPI) standard has been defined, and the first implementations of MPI have been reported[27]. Our group is working on an MPI implementation on top of Parix[20]. We will test the MPI implementation using the methods as described in this paper. Furthermore, the global communication routines of MPI, such as the *MPI_BCAST*, will be implemented such that the reported scaling behaviour of comparable Express and PVM routines can be improved.

6. CONCLUSIONS

We have compared the Iserver–Occam, Parix, Express and PVM parallel programming environments on a Parsytec GCel, by a detailed analysis of the performance of a particular application. Our approach, in which we start with an application, isolates the basic (environment)-dependent building blocks which are analysed independently, and combining all information in a time complexity analysis, allows us to compare the environments on all relevant degrees of functionality. Together with demands for portability of the code, and development time (i.e. programmability), an overall judgement of the environments can be made.

In general, we observe that increasing portability and programmability, in going from Iserver–Occam, via Parix to Express and PVM, results in a degradation of, in particular, the communication capabilities. The global communication routine of Express and PVM that we tested has a very bad scaling behaviour which clearly shows up in the larger partitions. This results in poor scalability of the Express and PVM implementation. Fortunately, in real production situations, with large problem sizes, the application has an efficiency very close to one, and the run time is mainly determined by the floating-point performance. In that situation Parix and PVM are the fastest, but Express, offering portable and easily implementable code, also has very good performance.

Application of our heuristic to compare PowerPVM with Parix on a Parsytec PowerXplorer shows that point-to-point communication in PowerPVM is as good as Parix. However, the global communication in PowerPVM is still far from optimal; future implementations of PVM and MPI should strive to more optimised global routines. In the future we will apply our methodology to compare the currently emerging MPI implementations with underlying native environments.

ACKNOWLEDGEMENTS

We wish to acknowledge financial support from the Netherlands Organisation for Scientific Research, grant 810-410-04 1. Furthermore, we would like to thank P. Trenning, F. Ettema and B. Jansen of the faculty of Mathematics and Computer Science of the University of Amsterdam for their assistance in this work.

REFERENCES

1. O.A. McBryan, 'An overview of message passing environments', *Parallel Comput.*, **20**, 417–444 (1994).
2. *CS Tools Documentation Guide*, A technical overview, Meiko Ltd., 1989.
3. M.D. May, P.W. Thompson and P.H. Welch, *Networks, Routers and Transputers*, IOS Press, Amsterdam, Oxford, Burke, 1993.
4. F. Langhammer, 'Second generation and teraflops parallel computers', in M. Valero, E. Onate, M. Jane, J.L. Larriba, and B. Suárez (Eds.), *Parallel Computing and Transputer Applications*, pp. 62–79.
5. Parsytec, 'The PARIX programming environment', in A. Allen (Eds.), *Transputer Systems – ongoing research*, IOS Press, Amsterdam, Oxford, Washington, Tokyo, 1992, pp. 218–230.
6. J. Flower and A. Kolawa, 'A packet history of message passing systems', *Phys. Rep.*, **207**, 291–304.
7. J. Dongarra, G.A. Geist, R. Manchek and V.S. Sundaram, 'Integrated PVM framework supports heterogeneous network computing', *Comput. Phys.*, **7**, 166–175 (1993).

8. R. Hempel, A.J.G. Hey, O. McBryan and D.W. Walker, 'Special Issue: Message Passing Interfaces', *Parallel Comput.*, **20** (1994).
9. M. Weber, 'Workstation clusters: one way to parallel computing', *Int. J. Mod. Phys. C*, **4**, 1307–1314 (1993).
10. A.R. Larrabee, 'The P4 parallel programming system, the Linda environment, and some experiences with parallel computing', *Sci. Program.*, **2**, 23–35 (1993).
11. A. Matrone, P. Schiano and V. Puotti, 'LINDA and PVM: A comparison between two environments for parallel programming', *Parallel Comput.*, **19**, 949–957 (1993).
12. T.G. Mattson, C.C. Douglas and M.H. Schultz, 'A comparison of CPS, Linda, P4, Posbyl, PVM, and TCGMSG: two node communication times', Tech. Rept. YALEU/DCS/TR-975 Dept. of Computer Science, Yale University, 1993.
13. C.C. Douglas, T.G. Mattson and M.H. Schultz, 'Parallel programming systems for workstation clusters', Tech. Rept. YALEU/DCS/TR-975, Dept. Computer Science, Yale University, 1993.
14. A.G. Hoekstra, 'Computer simulations of elastic light scattering:implementation and applications', Ph.D. Thesis, University of Amsterdam, 1994.
15. A.G. Hoekstra and P.M.A. Sloot, 'New computational techniques to simulate light scattering from arbitrary particles', in M. Maeda (Ed.), *Proceedings of the 3rd International Congress on Optical Particle Sizing '93 - Yokohama*, 1993, pp. 167–172.
16. A.G. Hoekstra and P.M.A. Sloot, 'Simulating elastic light scattering using high performance computing techniques', in A. Verbraeck and E.J.H. Kerckhoffs (Eds.), *European Simulation Symposium 1993*, Society for Computer Simulation International, 1993, pp. 462–470.
17. V. Balasundaram, G. Fox, K. Kennedy and U. Kremer, *A Static Performance Estimator in the Fortran D System*, Elsevier Science Publishers B.V., 1992.
18. J. Flower and A. Kolawa, 'Express is not just a message passing system: Current and future directions in Express', *Parallel Comput.*, **20**, 597–614 (1994).
19. G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon and D. Walker, *Solving Problems on Concurrent Processors 1: General Techniques and Regular Problems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
20. P.M.A. Sloot, Private communication, for more information you can send email to peterslo@fwi.uva.nl.
21. V.S. Sunderam, G.A. Geist, J. Dongarra and R. Manchek, 'The PVM concurrent computing system: Evolution, experiences, and trends', *Parallel Comput.*, **20**, 531–546 (1994).
22. V.S. Sunderam, 'PVM: A framework for parallel distributed computing', *Concurrency: Pract. Exp.*, **2**, 315–339 (1990).
23. E.M. Purcell and C.R. Pennypacker, 'Scattering and absorption of light by nonspherical dielectric grains', *Astrophys. J.*, **186**, 705–714 (1973).
24. S.F. Ashby, T.A. Manteuffel and P.E. Saylor, 'A taxonomy for conjugate gradient methods', *Siam J. Numer. Anal.*, **27**, 1542–1568 (1990).
25. A.G. Hoekstra, P.M.A. Sloot, W. Hoffmann and L.O. Hertzberger, 'Time complexity of a parallel conjugate gradient solver for light scattering simulations: theory and SPMD implementation'. Tech. Rept. CS-92-06, Faculty of Mathematics and Computer Science, University of Amsterdam, 1992.
26. W.H. Press, B.P. Flannery, S.A. Teukolsky and W.T. Vetterling, *Numerical Recipes in C, the Art of Scientific Computing*, Cambridge University Press, Cambridge, New York, Port Chester, Melbourne, Sydney, 1988.
27. D.W. Walker, 'The design of a standard message passing interface for distributed memory concurrent computers', *Parallel Comput.*, **20**, 657–673 (1994).