SCE based Parallel Processing and Applications in Simulation

René Fink, Sven Pawletta, Thorsten Pawletta, Wismar University, Germany
WWW.MB.HS-WISMAR.DE/cea
Bernhard Lampe, University of Rostock, Germany; bernhard.lampe@uni-rostock.de

In this paper, an overview of SCE based parallel processing is presented, and a new taxonomy for this field is discussed. More than 30 SCE based parallel processing projects are listed and categorized with respect to the new taxonomy. For the Multi-SCE class, performance parameters of several packages are presented in terms of latency and bandwidth. Furthermore, characterization schemes of parallel simulation and optimization applications are presented as well as performance results and characteristics of selected applications being parallelized under usage of Multi-SCE packages.

Introduction
Nowadays, Scientific and Technical Computing Environments (SCEs) like MATLAB, Scilab or Octave are well established in numerical computations. Particularly simulation and optimization applications are well supported by these environments, often by specialized subsystems and toolboxes. Before the introduction of SCEs, the development of scientific and technical computations was exclusively compiler based, so the process of programming, compiling, linking and testing had to go through several time consuming iterations. Advantages of SCEs in contrast to compiler based programming techniques are the way of interactive working, integrated numerical and visualization libraries, extendability and a very high level programming language.

Due to these advantages, rapid software prototyping was enabled in the field of computational science and engineering. One major drawback of SCE based programming is that interactive programming is based on an interpreter, so program execution in SCEs is significantly slower than program execution of compiled programs. To weaken this drawback, several possibilities exist, e.g. code optimization, compilation or parallel processing.

SCE based parallel processing was first investigated by The MathWorks Inc. in the middle of the 1980’s. These investigations showed disappointing performance results, published in 1995 ([1]). Beginning in the same year, first results of other research projects were published ([2], [3]), offering better performance. In a period of more than eleven years until today, several research and non-research projects concerning SCE based parallel processing were executed and produced a number of extensions to several SCEs, mostly MATLAB. In 2004, the MathWorks Inc. released the Distributed Computing Toolbox for Matlab ([4]), following the increasing demand for parallel processing in SCEs.

In an ongoing research project at the University of Applied Sciences Wismar, SCE based parallel processing techniques are investigated ([*]). Main motivations for this research are simulation and optimization problems, which cause extensive computations. Essential results of up to now investigations are presented in this paper. Section 1 introduces parallel processing, focusing on a basic taxonomy, recent parallel hardware architectures and programming techniques. Section 2 gives an overview of SCEs, including history, representatives, characteristics and possibilities of program acceleration. In Section 3, SCE based parallel processing is discussed. In this section, a new taxonomy on SCE based parallel processing is presented, followed by the characterization of existing projects and their assignment to this taxonomy. From the formerly presented taxonomy, Section 4 focuses the Multi-SCE class. In this section, results of Multi-SCE communication performance analysis are presented. Furthermore, new Multi-SCE prototypes are introduced and compared with existing Multi-SCE projects regarding communication performance. In Section 5, characterization schemes for simulation and optimization applications with respect to parallel processing are presented. Aspects of characterization are parallelism level, granularity and programming model applicability. In Section 6, simulation and optimization applications are presented, which have been parallelized under usage of Multi-SCEs. This presentation includes application characteristics as well as parallel runtime results. In Section 7, major facts of preceding sections are summarized.

1 Parallel Architectures and Programming Models

1.1 Basic Taxonomy

In the field of parallel processing a large number of taxonomies have been developed. Usually, taxonomy scopes are limited either to hardware or software.
In this article, a common taxonomy for both hardware and software structures is used. It is based on a conjunction of a very general interpretation of the well-known Flynn taxonomy ([5]) and the usual classification of memory structures into shared and distributed units. Flynn distinguishes hardware architectures by the number of instruction streams simultaneously applied to one or more data streams via processing elements, leading to the four basic classes SISD, SIMD, MISD and MIMD. In [6] it is shown that this principle is also applicable to software structures. For instance, a process or thread can be seen as processing element, where a code sequence is applied as instruction stream to a data stream. In this sense, Flynn’s classes can be described and interpreted as follows. In the SISD class, a single instruction stream (SI) is applied to a single data stream (SD) by one processing element, resulting in completely sequential processing. The SISD class contains no parallelism, but shows the delimitation of sequential structures to parallel structures. In the SIMD class, a single instruction stream is applied to multiple data streams by multiple processing elements. Since only one instruction stream exists, processing elements must work synchronously (synchronous parallelism). In the MISD class, multiple instruction streams are applied to a single data stream by multiple processing elements. In the MIMD class, multiple instruction streams are applied to multiple data streams using multiple processing elements. Since there are multiple independent instruction streams, processing elements do not have to work synchronously (asynchronous parallelism).

Parallel processing hardware is usually classified into shared and distributed memory architecture classes (SHM and DM). On a more abstract level, these concepts can be used for further refinement of the above general interpretation of Flynn’s taxonomy. In this sense, the SHM class describes a shared memory structure, where multiple processing elements have access to a common memory space. In contrast, the DM class describes a distributed memory structure with completely individual memory spaces for each processing element.

1.2 Hardware Architectures
The classification following Flynn’s taxonomy combined with memory structures is a common theoretical entrance to characterize parallel hardware architectures. But with respect to real hardware, not all classes of Flynn’s taxonomy are actually relevant today. The SISD hardware class has no relevance for parallel processing since it contains no parallelism at all.

The MISP hardware class is typically considered to be empty because no hardware structures exist following this processing scheme, though Flynn disagrees with that. The SIMD hardware class was important for large-scale parallel processing until the 1990’s, but disappeared almost completely today in this area. Thus, MIMD is the only class that is left over. Actually, a large variety of MIMD hardware exists and can be further classified with respect to memory structures.

Regarding memory structure, real MIMD architectures can be divided into three subclasses: shared, distributed and hybrid memory. Thereby, the memory structure strongly influences the scalability and communication performance of an architecture. Generally, distributed memory architectures are well scalable, but show low communication performance. On the other hand, shared memory architectures have limited scalability, but high communication performance. In hybrid memory architectures, shared memory structures on lower hierarchy levels are combined with distributed structures on higher levels. Therefore, hybrid memory architectures are well scalable and show high communication performance on lower hierarchy levels.

In a hardware architectures investigation of SNE Comparison CP1 contributions from 1994 until 2003 ([7]), being intended to attain a hardware overview in small scale parallel processing (especially in simulation domains), three types of hardware have been identified: Cluster Computers, Symmetric Multiprocessors and Transputers. Cluster Computers consist of independent processors with own physical memory, connected by a standard interconnect. They can be categorized as distributed memory MIMD systems. Symmetric Multiprocessors (SMP) consist of multiple homogeneous processors accessing a common physical memory. They can be classified as shared memory MIMD systems. Transputers consist of multiple processors with own physical memory, containing additional supporting hardware for fast interconnection. They can be classified as distributed memory MIMD systems. Transputers were well-established in small-scale parallel computing during the 1990’s, but disappeared almost completely today.

In a second hardware architectures investigation of systems appearing on the recent Top500 list, showing the 500 most performing parallel computers, also three types of hardware have been identified (www.top500.org): Cluster Computers, Constellations and Massively Parallel Computers.
Constellations consist of multiple SMPs, connected by a standard interconnect. Constellations can be classified as hybrid memory MIMD systems. Massively Parallel Computers consist of a very large number of processors (>1,000) with partly own and common physical memory, connected by a fast interconnect. They can be classified as distributed or hybrid memory MIMD systems.

The two investigations of parallel processing hardware show that recently only MIMD systems have practical relevance, both in small scale and large scale parallel processing. Hardware architecture differences regarding the described taxonomy can only be found in memory structures. Small scale parallel processing is dominated by distributed and shared memory architectures. In large scale parallel processing distributed and hybrid memory architectures are dominant. Shared memory systems are no longer present in the Top500 list because of their limited scalability. One can assume that hybrid memory architectures will also prevail in small-scale parallel processing in the near future. The main reason for this assumption is the establishment of multi core CPUs as commodity off-the-shelf (COTS) components, enabling installations of reasonable small scale Constellations.

1.3 Parallel Programming

As an interface between the programmer and the underlying parallel hardware structures, parallel programming models are used. Parallel programming models are supported either on the level of programming languages, compilers or applied libraries. Following Skillicorn ([8]), the development of parallel programs requires the fulfillment of four tasks:

- **Partitioning**: divide a problem into multiple partial problems
- **Instantiation and Mapping**: process startup and assignment to processors
- **Communication**: data exchange between processes
- **Synchronization**: management of process states

In a parallel programming model, these programming tasks can be explicit or implicit (visible or invisible in the program code). The distinction of parallel programming models is based on their set of explicit and implicit programming tasks. Programming models with only implicit programming tasks are referred to as implicit models, while programming models requiring at least one explicit task are referred to as explicit models.

Examples of implicit programming models are data parallel languages like FORTRAN 90, parallelizing compilers like Intel or SGI compiler suites and the application of parallel numerical libraries like ScALAPACK. Widespread models for explicit parallel programming are:

- Shared Memory Programming
- Message Passing Programming
- Remote Procedure Call (RPC) Programming

In all of these models, partitioning must be done explicitly, while explicitity of instantiation and mapping is system dependent. The explicit or implicit representation of synchronization and communication tasks is the main distinctive feature of the presented models, as shown in Table 1.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Memory</td>
<td>explicit</td>
<td>explicit or implicit</td>
<td>implicit</td>
<td>explicit</td>
</tr>
<tr>
<td>Message Passing</td>
<td>explicit</td>
<td>explicit or implicit</td>
<td>explicit</td>
<td>implicit</td>
</tr>
<tr>
<td>RPC</td>
<td>explicit</td>
<td>explicit or implicit</td>
<td>implicit</td>
<td>implicit</td>
</tr>
</tbody>
</table>

Table 1: Programming tasks in explicit parallel programming models.

In **Shared Memory Programming**, process communication is done implicitly via a common memory address space. On the other hand, process synchronization has to be organized explicitly, in the simplest case to avoid concurrent write accesses to the same addresses. Common shared memory programming standards are OpenMP and POSIX threads.

In **Message Passing Programming**, process communication has to be implemented explicitly by send/receive operations. In contrast to shared memory programming, process synchronization is completely implicit due to the causality of message transfer (a receiving process blocks until a desired message is sent). Well-known standards and libraries for message passing programming are MPI and PVM.

Unlike shared memory and message passing programming, **RPC Programming** is not originated in parallel programming. Classical synchronous RPC is a usual model in distributed programming. It is based on a client-server structure, where one server process serves multiple client processes. To make RPC usable in parallel programming, this scheme must be inverted, resulting in multiple server processes serving one client exclusively.
In such a scenario, the client splits up one problem into multiple partial problems, subsequently processed by the servers. After finishing all partial problems, the results are collected by the client. This programming scheme is also referred to as embarrassingly parallel or task parallel.

When using RPC in parallel programming, the classical synchronous RPC semantic must be extended either to asynchronous scalar RPC (non-blocking RPC [9]) or synchronous vectorial RPC (multiple RPC at the same time [6]). In all RPC variants, communication takes place implicitly by parameter passing to and from the remote procedure.

Regarding the introduced basic taxonomy, shared memory and message passing programming can be seen as native programming models on top of physical shared and distributed memory architectures, respectively. But of course, arbitrary mapping of programming models to certain physical architectures is possible by intermediate layers. The RPC programming model has no direct counterpart on the hardware level, but can be mapped by shared memory and message passing programming as well.

2 Scientific and Technical Computing Enviroments

A Scientific and Technical Computing Environment is a software system that enables interactive numerical scientific and technical computations and visualizations. The term SCE has been introduced in [6] as an abbreviation of Scientific and Technical Computing and Visualization Environment.

Before the introduction of SCEs, scientific and technical computations were executed by compiler based program development. To develop a program, the process of coding, compiling, linking and testing had to be iterated several times. Furthermore, to visualize computational results, additional programs or libraries were necessary.

The first major event of SCE history took place in 1977, when Research Systems Inc. released IDL (Interactive Data Language), enabling interpretative data analysis and visualization. Also in the late 1970’s, Cleve Moler developed the first version of MATLAB, a matrix based interpretative programming system for linear algebra problems. In 1984, the first commercial version of MATLAB was released by The MathWorks Inc., focussing technical computations and visualizations on the upcoming PC. At the same time, further commercial SCEs like Ctrl-C, MatrixX and Gauss were released, partly derived from the first MATLAB version and with MATLAB - similar syntax.

During the 1990’s, MATLAB was growing to a proprietary quasi-standard in computational engineering. In the same decade, several non-commercial SCEs arose (e.g. Octave in 1993, Scilab in 1994), partly with MATLAB compatible syntax and function sets. Nevertheless, MATLAB remained a standard, which is revealed by the large set of MATLAB extensions provided by The MathWorks Inc. and third parties. Table 2 summarizes today’s available SCEs.

<table>
<thead>
<tr>
<th>SCE</th>
<th>Producer</th>
<th>Initial Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL</td>
<td>Research Systems</td>
<td>1977, commercial</td>
</tr>
<tr>
<td>Gauss</td>
<td>Adept Systems</td>
<td>1983, commercial</td>
</tr>
<tr>
<td>Matlab</td>
<td>The MathWorks</td>
<td>1984, commercial</td>
</tr>
<tr>
<td>O-Matrix</td>
<td>Harmonic Software</td>
<td>1992, commercial</td>
</tr>
<tr>
<td>Octave</td>
<td>J. W. Eaton</td>
<td>1993, non commercial</td>
</tr>
<tr>
<td>Scilab</td>
<td>INRIA</td>
<td>1994, non commercial</td>
</tr>
<tr>
<td>Rlab</td>
<td>I. Searle</td>
<td>1994, non commercial</td>
</tr>
<tr>
<td>Tela</td>
<td>P. Janhunen</td>
<td>1994, non commercial</td>
</tr>
<tr>
<td>Euler</td>
<td>R. Grothmann</td>
<td>1996, non commercial</td>
</tr>
<tr>
<td>Yorick</td>
<td>D. H. Munro</td>
<td>1996, non commercial</td>
</tr>
</tbody>
</table>

Table 2: Today’s available SCEs, producers and initial release date.

2.1 SCE Characteristics

The following properties denote software systems as SCEs:

- Interactive way of working
- Integrated numerical libraries
- Integrated visualization libraries
- Extendable function set
- Matrix oriented high level programming language

Interactive way of working allows immediate data analysis as well as tests of new code lines or modules. Integrated numerical libraries provide users immediate access to well tested algorithms, while visualization libraries support instant visual data analysis. The extendable function set enables the addition of user defined functions or third party products to the SCE. The SCE’s high-level language allows variables to be defined, redefined or resized at any point in a program. The data type of variables is determined implicitly and can change during program execution. Basic data types are double precision matrices, extendable to complex or non-complex multidimensional arrays. Consequently, operators and integrated functions are array based, which results in compact data parallel code.

Figure 1 shows the basic structure of SCEs: data processing follows the classical von Neumann (SISD) scheme, processing one instruction stream over one data stream.
For this purpose, SCEs contain data stream input and output modules as well as an instruction stream input module represented by the interpreter. A memory management module storing internal data and a computational module containing internal and external function sets are further essential SCE components.

2.2 Acceleration of SCE Program Execution

As already mentioned, the usage of SCEs for scientific and technical program development takes considerably less time compared to compiler-based development. On the other hand, program execution in SCEs is significantly slower than execution of compiled programs. SCE code acceleration offers possibilities to weaken this drawback.

To accelerate SCE programs, profilers are useful. A profiler allows time analysis of every code line and simplifies identification of time-consuming program structures. Today, profilers are components of commercial SCEs like MATLAB, IDL or Gauss, only.

Several approaches to accelerate SCE program execution exist, also in combination with each other:

- Program optimization
- Compilation of SCE code
- Re-implementation in compilable language
- Parallel processing of SCE code

In case of program optimization, time-consuming code lines are substituted by more efficient program structures. Substitutions are for instance data pre-allocation instead of data resizing or data parallel operations instead of loops.

If compilation of SCE code is used, complete SCE programs or parts of them are automatically translated into compilable intermediate code and fed to a compiler subsequently (explicit compilation). Automatic compilation is also used by just-in-time compilers compiling program structures temporarily, so that time consuming interpretation is avoided (implicit compilation). Beside acceleration, compilation of SCE code is useful for SCE program deployment as a stand-alone application.

If re-implementation of SCE code is applied, time-consuming program structures are transformed manually into compilable code and compiled to machine code subsequently. Since prototype development already happened within the SCE, code re-implementation takes considerably less time than direct compiler based development. For the re-implementation approach, an interface between SCE and a compilable language is necessary. For instance, interfaces are provided by MATLAB (C, Fortran), Octave (C++) or Scilab (C, Fortran).

In parallel processing of SCE code, time-consuming program structures run in parallel on multiple processors. Discussions on SCE based parallel processing are topics of the following sections.

3 SCE - based Parallel Processing

3.1 Taxonomy

The combination of parallel processing and SCE based working can be realized in several ways. To ease comparisons of SCE based parallel processing variants, coarsely classifying taxonomies are proposed. In literature, two sources of taxonomies can be found. The first source ([6]) uses a SCE spanning taxonomy distinguishing four categories:

- Compilation Approach: compilation of SCE code into parallel machine code; execution on a parallel system
- Coupling Approach: coupling of SCE and parallel system; remote execution of parallel routines
- Parallel-SCE Approach: SCE runs on a parallel system; local execution of parallel routines
- Multi-SCE Approach: coupling of multiple SCEs, altogether representing a parallel system

The second source ([10]) uses a MATLAB based taxonomy and also distinguishes four categories:

- Embarrassingly Parallel: coupling of multiple MATLAB instances with one superordinated instance; RPC programming model.
- Message Passing: coupling multiple MATLAB instances without superordinated instances; message passing programming model.
- **Backend Support**: MATLAB as a front end to a parallel system executing parallel routines.
- **MATLAB Compiler**: compilation of MATLAB code into parallel machine code; execution on a parallel system.

A comparison of both taxonomies shows that in two cases categories of one taxonomy can be merged into one category of the other taxonomy. Firstly, Pawletta's **Coupling Approach** and **Parallel-SCE Approach** can be merged into Choy's **Backend Support**. Secondly, Choy's **Embarrassingly Parallel** category and **Message Passing** category can be merged into Pawletta's **Multi-SCE Approach**. Furthermore, the compiler-based category is the same in both taxonomies.

Therefore, both existing taxonomies can be brought together, resulting in a new taxonomy on SCE based parallel processing (see Figure 2):

- **Compilation Approach**: compilation of SCE code into parallel machine code; execution on a parallel system
- **Front End Approach**: SCE as a front end to a parallel system which executes parallel routines
- **Multi-SCE Approach**: coupling of multiple SCEs, altogether representing a parallel system

For the intermediate code, scalar languages like C or Fortran 77, and data parallel languages like Fortran 90 are used. Subsequently, the intermediate code is compiled by a parallelizing compiler or linked against parallel numerical libraries after compilation. To create a parallel program by compilation, no modification of SCE code is necessary, which indicates an implicit parallel programming model. For parallel execution of a compiled SCE program, no SCE is necessary. Therefore, SCEs are just utilized as development environments in this approach.

Table 3 summarizes projects following the compilation approach. Compiler based projects focusing on parallel signal processing hardware are neglected in this summarization due to their high specialization.

<table>
<thead>
<tr>
<th>Project</th>
<th>SCE</th>
<th>Intermediate Code</th>
<th>Rel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Falcon [12]</td>
<td>MATLAB</td>
<td>Fortran 90</td>
<td>1996</td>
</tr>
<tr>
<td>Menhir [13]</td>
<td>MATLAB</td>
<td>C or Fortran 77 incl. parallel linear algebra routines</td>
<td>1998</td>
</tr>
<tr>
<td>Otter [14]</td>
<td>MATLAB</td>
<td>C incl. parallel linear algebra or signal processing routines</td>
<td>1998</td>
</tr>
</tbody>
</table>

Figure 2: New taxonomy on SCE based parallel processing.

### 3.2 Compilation Approach

Following the compilation approach, sequential SCE programs or parts of them are translated into compilable intermediate code.

![Figure 2: New taxonomy on SCE based parallel processing.](image)

#### 3.3 Front End Approach

Using the front end approach, the SCE represents a user interface to a parallel processing platform. Within the SCE, users may call parallel routines interactively, often without necessity of special parallel computing knowledge. The parallel processing platform can be represented by a remote computer or a local parallel workstation. That means there is a distinction between remote and local SCE front ends. Since no modification of SCE code regarding parallel programming tasks is necessary (see Section 1.2), an implicit parallel programming model exists.
Using this approach, SCEs are development and runtime environments, but to run a parallel program, only a single SCE instance is necessary. For remote front ends, coupling follows the client server model. The client is represented by the SCE, while the remote parallel system acts as server. Servers provide mostly linear algebra routines like BLAS or SCALAPACK, but integration of user-defined routines is possible. Table 4 summarizes projects following the remote front end approach.

In the field of local front ends, the SCE’s computational module contains parallel numerical routines, executed on the local parallel computer or on a compound of local and remote computers. Parallel numerical standard routines as well as user-defined routines are possible. In the area of standard routines, internal SCE functions often use SIMD capabilities of modern CPUs like MMX or SSE extensions, for example in basic linear algebra, signal processing or trigonometry.

Standard routines which facilitate capabilities of MIMD structures like SMP or multi core CPUs are rarely integrated into SCEs. Currently, only IDL is known to facilitate such structures by its Multi-Threading libraries. User defined parallel routines can easily be integrated into the SCE’s computational module by usage of SCE interfaces to compilable languages (see Section 2.2).

### 3.4 Multi-SCE Approach

The Multi-SCE approach is characterized by multiple conventional SCE instances being connected by a coupling platform, altogether representing a Multi-SCE. The coupling platform consists of a physical layer like Ethernet, shared memory or disk units and of a software layer providing low level programming access to physical devices.

Suitable low-level services to build up Multi-SCEs are:
- **Operating System services** like Sockets, System V IPC, File I/O
- **External middleware services** like Java sockets, PVM, MPI, CORBA, DCOM, HLA
- **SCE internal middleware services** like Matlab engine

On top of these low level services a high level interface has to be realized within the Multi-SCE. On one hand, high level interfaces have to support a well-defined parallel programming model (see Section 1.3). On the other hand, the interface should be well integrated into the matrix oriented SCE programming paradigm (see Section 2.1). To fulfill both demands, it is necessary to adapt the classical programming models to the matrix-oriented paradigm.

In real Multi-SCEs the following adaptations are found:
- Shared Memory Prog. ↦ Shared Array Prog.
- Message Passing Prog. ↦ Array Passing Prog.
- asynchronous RPC ↦ vectorial RPC

Using the Multi-SCE approach, explicit parallel programming can be done in the SCE typical way of interactive working. In this approach, the whole SCE compound is both development and runtime environment. Hence, parallel program execution requires multiple SCE instances.

Due to its structure, the Multi-SCE approach is well suited for cluster and grid platforms and therefore attractive to a broad user community. This fact is reflected by the large number of Multi-SCE packages released during the last decade, being listed in Table 5. Since some Multi-SCE packages do not adapt parallel programming models in the way described above, programming models in Table 5 are classified by general terms, i.e. Message Passing (MP), Shared Memory (SHM) and RPC.

In [6] also combinations of SCE based parallel processing approaches have been proposed. The usage of such hybrid approaches offers ways to use the Multi-SCE approach only in development phases, while switching to the front end approach in production phases. On the other hand, hierarchies like front end approach on lower levels and Multi-SCE approach on higher levels are imaginable. Up to now, hybrid approaches have not been investigated in detail.

### 4 Multi-SCE Performance Analysis

To investigate Multi-SCE packages quantitatively, communication performance measurements have been executed. The aim of measurements applied is the ascertainment of comparable Multi-SCE spanning performance parameters.
With this method, a master-slave program structure has been used, where the slave process reflects messages issued by the master process. To determine average round trip times, the send/receive process is repeated several times, altogether embedded into time measurement in the master process. The method described is well suited for packages using a message passing programming model. For RPC and shared memory models, this method has to be adapted to deliver comparable results.

In RPC programming, time measurements have been executed using a dummy function on the slave side, which just returns a single input parameter. On the master side, the dummy function is called remotely with a message as input parameter, resulting in the same message transfer as described above.

In shared memory programming, explicit synchronization has been applied. In the master process, a shared variable read. The slave process behaves analogically, so message transfer follows the scheme described above.

Round trip time measurements have been executed with different data sizes, resulting in a dependence of round trip time to data size. If round trip time is linearly dependent on message size, which is the case in executed investigations, linear regression of round trip time at different data sizes leads to communication performance significant parameters byte rate and latency. The following formula describes the dependencies:

\[ t_{RoundTrip} = 2 \cdot \frac{DataSize}{ByteRate} + t_{Latency} \]

As measurement platform, two Linux computers running with kernel version 2.4.18 have been used. Computers are driven by a 1,500 MHz AMD Athlon CPU and are connected by Gigabit Ethernet.

### 4.2 Performance Results of Existing Multi-SCE Packages

Out of all 24 listed Multi-SCE packages in Table 5, only 8 packages could be investigated regarding communication performance. The remaining 16 packages were either outdated (not adapted to recent SCEs) or not available. Results for investigated Multi-SCE packages are summarized in Table 6.

The results presented show that communication performance of Multi-SCEs heavily depends on the used low-level service but weakly depends on the applied SCE.

<table>
<thead>
<tr>
<th>Project</th>
<th>SCE</th>
<th>Low Level Service</th>
<th>Prog. Mod.</th>
<th>Rel.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT Toolbox [17]</td>
<td>MATLAB</td>
<td>external (PVM)</td>
<td>MP</td>
<td>1995</td>
</tr>
<tr>
<td>PVM Tbx. [19]</td>
<td>Scilab</td>
<td>external (PVM)</td>
<td>MP</td>
<td>1998</td>
</tr>
<tr>
<td>PMI [20]</td>
<td>MATLAB</td>
<td>internal (engine)</td>
<td>RPC</td>
<td>1999</td>
</tr>
<tr>
<td>Matmarks [21]</td>
<td>MATLAB</td>
<td>external (ThreadMarks)</td>
<td>SHM</td>
<td>1999</td>
</tr>
<tr>
<td>PVM Tbx. [22]</td>
<td>MATLAB</td>
<td>external (PVM)</td>
<td>MP</td>
<td>1999</td>
</tr>
<tr>
<td>Cornell MT Tbx. [23]</td>
<td>MATLAB</td>
<td>external (MPI)</td>
<td>MP</td>
<td>2000</td>
</tr>
<tr>
<td>PLab [24]</td>
<td>MATLAB</td>
<td>OS (Sockets)</td>
<td>RPC</td>
<td>2000</td>
</tr>
<tr>
<td>ParMatlab [25]</td>
<td>MATLAB</td>
<td>OS (Sockets)</td>
<td>RPC</td>
<td>2001</td>
</tr>
<tr>
<td>MPI Toolbox [22]</td>
<td>MATLAB</td>
<td>external (MPI)</td>
<td>MP</td>
<td>2001</td>
</tr>
<tr>
<td>IDL/PVM [26]</td>
<td>IDL</td>
<td>external (PVM)</td>
<td>MP</td>
<td>2001</td>
</tr>
<tr>
<td>MatlabMPI [27]</td>
<td>MATLAB</td>
<td>OS (File I/O)</td>
<td>MP</td>
<td>2001</td>
</tr>
<tr>
<td>Beolab Tbx. [28]</td>
<td>MATLAB</td>
<td>internal (engine)</td>
<td>RPC</td>
<td>2002</td>
</tr>
<tr>
<td>Parallelization Tk. [29]</td>
<td>MATLAB</td>
<td>internal (engine)</td>
<td>RPC</td>
<td>2002</td>
</tr>
<tr>
<td>DistributedPP [30]</td>
<td>MATLAB</td>
<td>OS (File I/O)</td>
<td>RPC</td>
<td>2002</td>
</tr>
<tr>
<td>MPIDL [31]</td>
<td>IDL</td>
<td>external (MPI)</td>
<td>MP</td>
<td>2003</td>
</tr>
<tr>
<td>Parallel [32]</td>
<td>Gauss</td>
<td>external (Java sockets)</td>
<td>RPC</td>
<td>2003</td>
</tr>
<tr>
<td>Parallel Octave [33]</td>
<td>Octave</td>
<td>external (MPI)</td>
<td>MP</td>
<td>2003</td>
</tr>
<tr>
<td>Distributed Octave [34]</td>
<td>Octave</td>
<td>external (MPI)</td>
<td>RPC</td>
<td>2004</td>
</tr>
<tr>
<td>MPI Toolbox [35]</td>
<td>Octave</td>
<td>external (MPI)</td>
<td>MP</td>
<td>2004</td>
</tr>
<tr>
<td>DC Toolbox [36]</td>
<td>MATLAB</td>
<td>external (Java Jini, MPI)</td>
<td>MP, RPC</td>
<td>2004</td>
</tr>
<tr>
<td>pMatlab [37]</td>
<td>MATLAB</td>
<td>OS (File I/O)</td>
<td>SHM</td>
<td>2005</td>
</tr>
<tr>
<td>MDICE [38]</td>
<td>MATLAB</td>
<td>OS (Sockets)</td>
<td>RPC</td>
<td>2005</td>
</tr>
<tr>
<td>Matlab2Matlab [39]</td>
<td>MATLAB</td>
<td>external (Java sockets)</td>
<td>RPC</td>
<td>2005</td>
</tr>
<tr>
<td>GAMMA [40]</td>
<td>MATLAB</td>
<td>external (Global arrays)</td>
<td>SHM</td>
<td>2006</td>
</tr>
</tbody>
</table>

Table 5: Projects following the Multi-SCE approach.

For parameter ascertainment, a programming model independent measurement method has been developed and applied. Measurements have been executed both with existing Multi-SCE packages and new Multi-SCE prototypes.

### 4.1 Measurement Method and Platform

Communication performance measurements have been applied using a round trip time (RTT) method with a one-to-one communication scheme.
So byte rates of packages using PVM (DP Toolbox and PVM/Scilab) are nearly the same although SCEs differ. The same effect can be seen on packages using LAM MPI. Furthermore, packages that do not use message passing libraries show significant higher latencies than message passing library based packages.

### 4.3 Performance Results of New Multi-SCE Prototypes

For future designs of Multi-SCE packages, new Multi-SCE prototypes have been developed using MATLAB as targeted SCE. Aims of these prototype developments are feasibility studies and performance improvements compared to existing packages.

Up to now, five prototypes have proofed to be feasible (communication performance results of prototypes are shown in Table 7):

- Fast PVM prototype
- MPICH2 prototype
- Shared Arrays on top of MPICH2 prototype
- Matlab engine plus threads prototype
- Java sockets based Message Passing prototype

The fast PVM prototype is a component of the latest DP Toolbox release (v1.7.2), which has been entirely rewritten.

It shows a slightly higher byte rate and lower latency than the interface used in DP v1.5.

The MPICH2 prototype is the first freely available MATLAB toolbox facilitating the MPICH2 library. Prototypes using other MPI libraries (e.g. LAM or MPICH) have shown to be unstable with MATLAB. Compared to Table 6, the MPICH2 prototype shows very low latency.

The Shared Arrays prototype is built on top of the MPICH2 prototype. It is intended to investigate mappings of parallel programming models, especially from message passing to shared memory programming. Since a distributed memory process model is the basis of this prototype, shared memory can only be virtual. In this prototype, a server process manages all shared variables, while shared memory accessing processes act as clients, issuing put memory and get memory commands. Compared to the MPICH2 prototype, which uses the same low-level service, this prototype shows lower byte rate and higher latency. This is caused by frequent send/receive commands on client/server communication as well as by additional barrier synchronization effort.

The thread using MATLAB engine prototype represents a new approach to control multiple Matlab Engines simultaneously. Matlab engine is a library that allows the invocation and control of a remote MATLAB instance out of a C program. To execute a remote command within the MATLAB instance, the blocking function `engEvalString()` has to be called. To use MATLAB engines in parallel computing, this blocking behavior needs to be circumvented. Existing Multi-SCE packages that use Matlab engines bypass blocking by misusing engine internal functionalities (write/read on engine internal file pointers, not working on Win32 MATLAB), introduced by PMI.

<table>
<thead>
<tr>
<th>Package, SCE</th>
<th>Low Level Service</th>
<th>Prog. Mod.</th>
<th>Byte Rate [MB/s]</th>
<th>Latency [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP Tbx. v1.5, MATLAB v7.1</td>
<td>PVM v3.4</td>
<td>MP</td>
<td>11.9</td>
<td>1.1</td>
</tr>
<tr>
<td>PVM Tbx., Scilab v2.7</td>
<td>PVM v3.4</td>
<td>MP</td>
<td>12.0</td>
<td>0.2</td>
</tr>
<tr>
<td>MPI Tbx., MATLAB v7.1</td>
<td>LAM v7.1 (MPI)</td>
<td>MP</td>
<td>36.7</td>
<td>0.1</td>
</tr>
<tr>
<td>MPI Tbx., Octave v2.1</td>
<td>LAM v7.1 (MPI)</td>
<td>MP</td>
<td>37.2</td>
<td>0.1</td>
</tr>
<tr>
<td>DC Tbx. v2.0, MATLAB v7.1</td>
<td>Jini</td>
<td>RPC</td>
<td>3.6</td>
<td>421.9</td>
</tr>
<tr>
<td>MATLAB MPI v1.2, MATLAB v7.1</td>
<td>MPICH2 v1.0</td>
<td>MP</td>
<td>34.1</td>
<td>0.3</td>
</tr>
<tr>
<td>Beolab Tbx., MATLAB v7.1</td>
<td>Matlab engine, RPC</td>
<td>MP</td>
<td>25.5</td>
<td>44.8</td>
</tr>
<tr>
<td>Parallelization Tk. v1.2, MATLAB v7.1</td>
<td>Matlab engine, RPC</td>
<td>MP</td>
<td>37.1</td>
<td>81.8</td>
</tr>
</tbody>
</table>

### Table 6: Communication performance of existing Multi-SCE packages.

<table>
<thead>
<tr>
<th>Prototype</th>
<th>Low Level Service</th>
<th>Prog. Mod.</th>
<th>Byte Rate [MB/s]</th>
<th>Latency [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fast PVM</td>
<td>PVM v3.4</td>
<td>MP</td>
<td>13.3</td>
<td>0.3</td>
</tr>
<tr>
<td>MPICH2</td>
<td>MPICH2 v1.0</td>
<td>MP</td>
<td>29.9</td>
<td>0.1</td>
</tr>
<tr>
<td>Shared Arrays</td>
<td>MPICH2 v1.0</td>
<td>SHM</td>
<td>11.0</td>
<td>1.7</td>
</tr>
<tr>
<td>Engine plus threads</td>
<td>Matlab engine, threads</td>
<td>RPC</td>
<td>31.0</td>
<td>20.9</td>
</tr>
<tr>
<td>Java MP</td>
<td>Java sockets, threads</td>
<td>MP</td>
<td>1.4</td>
<td>2.4</td>
</tr>
</tbody>
</table>

### Table 7: Communication performance of Multi-SCE prototypes (SCE: Matlab v7.1).
The thread using approach encapsulates all MATLAB engine calls into threads, leading to an engine implementation independent interface, working also with Win32 MATLAB. In comparison with other engine based packages, this prototype offers the lowest latency. The Java based message passing package is a prototype of a platform independent message passing library, working with all MATLAB architectures (e.g. Win32, Linux, Mac, Solaris). Inside this library, TCP Socket accesses are encapsulated within threads leading to a communication that follows the message passing paradigm. This approach disappoints with the lowest byte rate out of all prototypes.

5 Characteristic of Simulation Applications

Since simulation applications are well supported in SCEs, for instance by ODE and PDE solvers or simulation subsystems like Simulink, Stateflow or Scicos, they are focused in application based investigations. Furthermore, simulation experiments and simulation based optimizations belong to the most time consuming applications in SCEs today.

To characterize investigated applications with respect to parallel processing, three qualitative criteria are used:
- Parallelism level
- Granularity
- Programming model applicability

5.1 Parallelism Level

From a programmer's point of view, simulation based optimization is accomplished at several program levels. On the highest level, an optimization method tries to find optimal input parameters of an objective function that has to be minimized. Within the optimization method, objective functions are called to calculate objective values for current input parameter sets. Within the objective function, results of simulation runs are used to calculate objective values.

Following this view, parallelism in simulation and optimization can occur on the described program levels, subsequently referred to as parallelism levels.

Parallelism levels can be categorized in the following way, ordered from highest to lowest program level:
- Independent optimization runs
- Parallel optimization methods
- Independent simulation runs
- Distributed models

Independent optimization runs are placed on the highest parallelism level. They occur, for instance if optimizations on multiple model operating points are necessary. In parallel optimization methods multiple objective values are calculated independently during one iteration step. Examples of parallel optimization methods are evolutionary strategies or Monte Carlo methods. Independent simulation runs do not have to be embedded into objective functions necessarily, but can also represent stand-alone simulation experiments. They occur, for instance in parameter studies. Distributed models are placed on the lowest parallelism level. At this level, a model is split up into partial models, subsequently simulated by multiple cooperative processes. In discrete event simulation, those processes are referred to as logical processes ([41]).

5.2 Granularity

In terms of parallel processing, granularity denotes the ratio of computation effort to communication effort in a parallel program. Distinctions are made between coarse, middle and fine granular programs. A program shows coarse granularity, if communication effort is negligible compared to computation effort. On the other hand, fine granularity exists, if communication effort dominates seriously over computation effort. Granularity represents a qualitative measure and is also hardware dependent. So programs that show middle granularity on one hardware, could be classified as fine granular on another hardware.

Regarding parallelism level, granularity tends to decrease from the highest to the lowest level.

5.3 Programming Model Applicability

Programming model applicability means the applicability of a parallel programming model to parallelize certain application structures. Since RPC programming is mappable onto shared memory and message passing programming (see sect. 1.3), every application being parallelizable by RPC is also parallelizable by shared memory or message passing programming.

Hence, a classification regarding program model applicability distinguishes two classes of applications:
- **RPC capable**: parallelizable by message passing, shared memory or RPC programming
- **Non RPC capable**: parallelizable only by message passing or shared memory programming
In simulation based applications, programming model applicability classification can be assigned unambiguously to parallelism levels, as shown in Table 8.

<table>
<thead>
<tr>
<th>Parallelism level</th>
<th>Programming model applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Independent optimization runs</td>
<td>RPC capable</td>
</tr>
<tr>
<td>Parallel optimization methods</td>
<td>RPC capable</td>
</tr>
<tr>
<td>Independent simulation runs</td>
<td>RPC capable</td>
</tr>
<tr>
<td>Distributed models</td>
<td>non RPC capable</td>
</tr>
</tbody>
</table>

Table 8: Relation of parallelism level and programming model applicability in simulation and optimization applications.

6 Application Performance Analysis

In this section, a number of optimization and simulation applications, being parallelized on basis of the Multi-SCE approach are discussed. Application based investigations comprise benchmark problems as well as real life applications from industry and research. In Table 9, applications are listed including their parallelism level and granularity degree. In every application, parallelization took place only on the highest possible parallelization level.

<table>
<thead>
<tr>
<th>Application</th>
<th>Parallelism Level</th>
<th>Granularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter optimization of an exhaust gas model</td>
<td>independent optimization runs</td>
<td>coarse</td>
</tr>
<tr>
<td>Evolutionary safety testing of embedded control software</td>
<td>parallel optimization method</td>
<td>coarse</td>
</tr>
<tr>
<td>Parameter study of a dynamic system</td>
<td>independent simulation runs</td>
<td>coarse</td>
</tr>
<tr>
<td>Simulation of Cavity Flow by the Lattice Boltzmann Method</td>
<td>distributed models</td>
<td>middle</td>
</tr>
<tr>
<td>Solution of a PDE by finite differences</td>
<td>distributed models</td>
<td>fine</td>
</tr>
<tr>
<td>Simulation of coupled ODE systems</td>
<td>distributed models</td>
<td>fine</td>
</tr>
</tbody>
</table>

Table 9: Parallelism level and granularity of investigated applications.

All investigations have been performed on a cluster computer, representing a distributed memory MIMD architecture. Cluster nodes are driven by a 1.500 MHz AMD Athlon CPU and are connected by Gigabit Ethernet. Operating system is GNU/Linux with kernel version 2.4.18.

The parameter optimization of an exhaust gas model represents a real life application from automotive industry. Typically, the optimization has to be performed for a large number of operating points of the same simulation model. Therefore, the parallelism level is independent optimization runs. With a number of 12 processors, a speedup of 10.3 has been reached on the test platform. As expected, this application shows the highest efficiency of 0.9 (efficiency = speedup divided by number of processors) among all selected applications, caused by parallelism level and granularity. A more detailed representation of this application can be found in [42].

The evolutionary safety testing of embedded control is also a real life application from automotive industry. In contrast to the first example it is a single optimization problem with embedded simulation. Nevertheless, the parallelism level is still in the optimization because an evolutionary search algorithm is used. Therefore, in each optimization step multiple simulation based objective values can be calculated simultaneously.

The general structure of this application is well suited for parallelization in principle. But the investigation of a certain application configuration has shown only a speedup of 4.7 with 12 processors on the test platform, leading to an efficiency of 0.4. The reason for this weak result is a relatively small-scaled embedded simulation in combination with very high latencies of the applied Multi-SCE package (see Table 10 and Table 6). This example illustrates that the relation of parallelism level and granularity is only of structural nature and does not guarantee good speedup values in each application case.

The parameter study of a dynamic system is a benchmark problem of the SNE Comparison CP1. In this example, a mass spring system has to be simulated with varying parameters, while system responses have to be averaged. Hence, the parallelism level is independent simulation runs. On the test platform, the investigated implementation has reached a speedup of 5.6 under usage of 8 processors, corresponding to an efficiency of 0.7. Some more details of the benchmark solution are published in [45].

The simulation of Cavity Flow by the Lattice Boltzmann Method is also a typical benchmark problem. In the example investigated, an incompressible fluid is bounded in a square cavity, driven by a uniform flow on the top boundary. Parallelization takes place by domain decomposition of the cavity in y-direction. In analogy to distributions in discrete event and continuous models, the applied decomposition can be seen as a model distribution approach.
Because of more or less strong dependencies between distributed model parts such applications are structurally not coarse grained. The investigated configuration has reached a speedup of 5.5 using 12 processors, corresponding an efficiency of 0.5.

The solution of a PDE by finite differences is a second benchmark problem from the SNE Comparison CP1. In this example, the motion of a swinging rope, described by a partial differential equation, is simulated under usage of the finite difference method. Similar to the above Lattice Boltzmann problem, parallelization takes place on the model level by domain decomposition of the rope. In this example, the computation effort in each distributed model part becomes very small compared to the necessary communication effort. Therefore, the application has not been speeded up on the test platform successfully. Actually, run time increases dramatically in comparison to the sequential solution. As in every non coarse grained application, the reachable speedup strongly depends on the run time platform applied. Therefore, also successful solutions of this benchmark have been published within the scope of SNE Comparison CP1.

The simulation of coupled ODE systems is a third benchmark problem from the SNE Comparison CP1. In this example, five coupled predator-prey systems are simulated. This benchmark problem is an example of the model distribution approach in continuous simulation. Due to the usually tight coupling between continuous model parts, a successful parallelization of such problems is mostly impossible on distributed memory MIMD systems. Therefore, also with this benchmark no speedup has been reached on the test platform.

Table 10 summarizes performance results of investigated simulation and optimization based application problems and shows the involved Multi-SCE packages.

<table>
<thead>
<tr>
<th>Application</th>
<th>Multi-SCE package</th>
<th>np</th>
<th>Speed up</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter optimization of an exhaust gas model [42]</td>
<td>DP Tbx. v1.5.0</td>
<td>12</td>
<td>10.3</td>
<td>0.9</td>
</tr>
<tr>
<td>Evolutionary safety testing of embedded control software [43]</td>
<td>DC Tbx. V2.0</td>
<td>12</td>
<td>4.7</td>
<td>0.4</td>
</tr>
<tr>
<td>Parameter study of a dynamic system [7]</td>
<td>DP Tbx. v1.5.0</td>
<td>8</td>
<td>5.6</td>
<td>0.7</td>
</tr>
<tr>
<td>Simulation of Cavity Flow by the Lattice Boltzmann Method [44]</td>
<td>DP Tbx. v1.7.0</td>
<td>12</td>
<td>5.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Solution of a PDE by finite differences [7]</td>
<td>DP Tbx. v1.5.0</td>
<td>8</td>
<td>&lt;1</td>
<td>-</td>
</tr>
<tr>
<td>Simulation of coupled ODE systems [7]</td>
<td>DP Tbx. v1.5.0</td>
<td>5</td>
<td>&lt;1</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 10: Performance of investigated applications (np: number of processors).

In such cases, message passing programming can be applied more efficiently, since there are no restrictions with respect to script or function execution and variable accesses.

7 Conclusion

Regarding parallel processing in general, it was shown that hardware specific classifications of Flynn and memory structure could be seen as basic taxonomies for hardware and software structures, as well.

The analysis of hardware structures in small scale and large scale parallel processing has demonstrated that in both domains MIMD systems dominate, so further distinctions with respect to memory structure are necessary.

It was assumed that in small scale parallel processing constellations consisting of multi core workstations will prevail in the near future. The discussion of programming models has shown that beside the two common explicit parallel programming models, shared memory and message passing programming, RPC programming is a convenient model for specific problems.

It was demonstrated that for the usage of RPC programming in parallel processing, certain adaptations to the classical RPC programming have to be made, namely asynchronous or vectorial RPC.
With respect to scientific and technical computing environments, typical denoting features of such systems have been presented. Besides SCE based parallel processing, several other ways of SCE program acceleration have been discussed.

Regarding SCE based parallel processing, a new taxonomy combining two existing taxonomies have been presented. Principles of all classes of the new taxonomy have been discussed and more than 30 representatives have been identified and categorized.

In the field of the Multi-SCE approach, a comparable communication performance measurement method for all parallel programming models was developed enabling comparisons on the basis of two parameters: latency and byte rate. Results of communication performance measurements of existing Multi-SCE packages have been presented. New Multi-SCE prototypes have been discussed and communication performance has been compared to existing packages.

With respect to applications, general characteristics of parallel simulation and optimization problems have been discussed. Especially, a parallelization level based scheme has been presented, allowing the characterization of both parallel optimization and simulation problems. Regarding certain applications being parallelized under usage of Multi-SCE packages, formerly introduced characteristics have been applied. Parallel performance results have been presented and discussed with respect to application's characteristics.

References


[*] The research is partly supported by BMBF and BM-M (HWP310-FHW13).

**Corresponding author:** René Fink
Rene Fink, Sven Pawletta, Thorsten Pawletta
Res. Group Computational Engineering and Automation, Wismar University, University of Technology, Business and Design, PF 1210, 23952 Wismar, Germany
WWW.MB.HS-WISMAR.DE/cea
Bernhard Lampe, Institute of Automation
University of Rostock, Richard-Wagner-Str. 31/H.8
18119 Rostock, Germany; bernhard.lampe@uni-rostock.de

Received: June 15, 2006
Revised: June 29, 2006
Accepted: July 20, 2006