Introduction to Parallel Computing

Bernd Mohr

published in

Multiscale Simulation Methods in Molecular Sciences,
J. Grotendorst, N. Attig, S. Blügel, D. Marx (Eds.),
Institute for Advanced Simulation, Forschungszentrum Jülich,
NIC Series, Vol. **42**, ISBN 978-3-9810843-8-2, pp. 535-549, 2009.

Introduction to Parallel Computing

Bernd Mohr

Institute for Advanced Simulation (IAS)
Jülich Supercomputing Centre (JSC)
Forschungszentrum Jülich, 52425 Jülich, Germany
E-mail: b.mohr@fz-juelich.de

The major parallel programming models for scalable parallel architectures are the message passing model and the shared memory model. This article outlines the main concepts of these models as well as the industry standard programming interfaces MPI and OpenMP. To exploit the potential performance of parallel computers, programs need to be carefully designed and tuned. We will discuss design decisions for good performance as well as programming tools that help the programmer in program tuning.

1 Introduction

Many applications like numerical simulations in industry and research as well as commercial applications such as query processing, data mining, and multi-media applications require more compute power than provided by sequential computers. Current hardware architectures offering high performance do not only exploit parallelism within a single processor via multiple CPU cores but also apply a medium to large number of processors concurrently to a single computation. High-end parallel computers currently (2009) deliver up to 1 Petaflop/s (10^{15} floating point operations per second) and are developed and exploited within the ASC (Advanced Simulation and Computing) program of the Department of Energy in the USA and PRACE (Partnership for Advanced Computing in Europe) in Europe. In addition, the current trend to multi-core processors also requires parallel programming to fully exploit the compute power of the multiple cores.

This article concentrates on programming numerical applications on parallel computer architectures introduced in Section 1.1. Parallelization of those applications centers around selecting a decomposition of the data domain onto the processors such that the workload is well balanced and the communication between processors is reduced (Section 1.2)[4].

The parallel implementation is then based on either the message passing or the shared memory model (Section 2). The standard programming interface for the message passing model is MPI (Message Passing Interface)[8–12], offering a complete set of communication routines (Section 3). OpenMP[13–15] is the standard for directive-based shared memory programming and will be introduced in Section 4.

Since parallel programs exploit multiple threads of control, debugging is even more complicated than for sequential programs. Section 5 outlines the main concepts of parallel debuggers and presents TotalView[21] and DDT[3], the most widely available debuggers for parallel programs.

Although the domain decomposition is key to good performance on parallel architectures, program efficiency also heavily depends on the implementation of the communication and synchronization required by the parallel algorithms and the implementation techniques chosen for sequential kernels. Optimizing those aspects is very system dependent and thus, an interactive tuning process consisting of measuring performance data and applying optimizations follows the initial coding of the application. The tuning process is supported by programming model specific performance analysis tools. Section 6 presents basic performance analysis techniques.

1.1 Parallel Architectures

A *parallel computer* or *multi-processor system* is a computer utilizing more than one processor. A common way to classify parallel computers is to distinguish them by the way how processors can access the system's main memory because this influences heavily the usage and programming of the system.

In a *distributed memory architecture* the system is composed out of single-processor nodes with local memory. The most important characteristic of this architecture is that access to the local memory is faster than to remote memory. It is the challenge for the programmer to assign data to the processors such that most of the data accessed during the computation are already in the node's local memory. Two major classes of distributed memory computers can be distinguished:

No Remote Memory Access (NORMA) computers do not have any special hardware support to access another node's local memory directly. The nodes are only connected through a computer network. Processors obtain data from remote memory only by exchanging messages over this network between processes on the requesting and the supplying node. Computers in this class are sometimes also called **Network Of Workstations (NOW)** or **Clusters Of Workstations (COW)**.

Remote Memory Access (RMA) computers allow to access remote memory via specialized operations implemented by hardware, however the hardware does not provide a global address space, i.e., a memory location is not determined via an address in a shared linear address space but via a tuple consisting of the processor number and the local address in the target processor's address space.

The major advantage of distributed memory systems is their ability to scale to a very large number of nodes. Today (2009), systems with more than 210,000 cores have been built. The disadvantage is that such systems are very hard to program.

In contrast, a *shared memory architecture* provides (in hardware) a global address space, i.e., all memory locations can be accessed via usual load and store operations. Access to a remote location results in a copy of the appropriate cache line in the processor's cache. Therefore, such a system is much easier to program. However, shared memory systems can only be scaled to moderate numbers of processors, typically 64 or 128. Shared memory systems are further classified according to the quality of the memory accesses:

Uniform Memory Access (UMA) computer systems feature one global shared memory subsystem which is connected to the processors through a central bus or memory switch. All of the memory is accessible to all processors in the same way. Such a system is also often called **Symmetrical Multi Processor (SMP)**.

Non Uniform Memory Access (NUMA) computers are more scalable by physically distributing the memory but still providing a hardware implemented global address space. Therefore access to memory local or close to a processor is faster than to remote memory. If such a system has additional hardware which also ensures that multiple copies of data stored in different cache lines of the processors is kept coherent, i.e., the copies always do have the same value, then it is called a **Cache-Coherent Non Uniform Memory Access (ccNUMA)** system. ccNUMA systems offer the abstraction of a shared linear address space resembling physically shared memory systems. This abstraction simplifies the task of program development but does not necessarily facilitate program tuning.

While most of the early parallel computers were simple single processor NORMA systems, today's large parallel systems are typically *hybrid systems*, i.e., shared memory NUMA nodes with a moderate number of processors are connected together to form a distributed memory cluster system.

1.2 Data Parallel Programming

Applications that scale to a large number of processors usually perform computations on large data domains. For example, crash simulations are based on partial differential equations that are solved on a large finite element grid and molecular dynamics applications simulate the behavior of a large number of particles. Other parallel applications apply linear algebra operations to large vectors and matrices. The elemental operations on each object in the data domain can be executed in parallel by the available processors.

The scheduling of operations to processors is determined by a *domain decomposition*[5] specified by the programmer. Processors execute those operations that determine new values for elements stored in local memory (owner-computes rule). While processors execute an operation, they may need values from other processors. The domain decomposition has thus to be chosen so that the distribution of operations is balanced and the communication is minimized. The third goal is to optimize single node computation, i.e., to be able to exploit the processor's pipelines and the processor's caches efficiently.

A good example for the design decisions taken when selecting a domain decomposition is Gaussian elimination[1]. The main structure of the matrix during the steps of the algorithm is outlined in Figure 1.

The goal of this algorithm is to eliminate all entries in the matrix below the main diagonal. It starts at the top diagonal element and subtracts multiples of the first row from the second and subsequent rows to end up with zeros in the first column. This operation is repeated for all the rows. In later stages of the algorithm the actual computations have to be done on rectangular sections of decreasing size. If the main diagonal element of the current row is zero, a pivot operation has to be performed. The subsequent row with the maximum value in this column is selected and exchanged with the current row.

A possible distribution of the matrix is to decompose its columns into blocks, one block for each processor. The elimination of the entries in the lower triangle can then be performed in parallel where each processor computes new values for its columns only. The main disadvantage of this distribution is that in later computations of the algorithm only a subgroup of the processors is actually doing any useful work since the computed rectangle is getting smaller.

To improve load balancing, a cyclic column distribution can be applied. The computations in each step of the algorithm executed by the processors differ only in one column.

In addition to load balancing also communication needs to be minimized. Communication occurs in this algorithm for broadcasting the current column to all the processors since it is needed to compute the multiplication factor for the row. If the domain decomposition

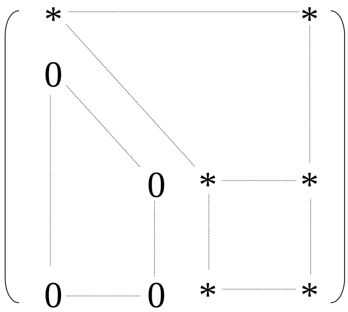

Figure 1. Structure of the matrix during Gaussian elimination.

is a row distribution, which eliminates the need to communicate the current column, the current row needs to be broadcast to the other processors.

If we consider also the pivot operation, communication is necessary to select the best row when a row-wise distribution is applied since the computation of the global maximum in that column requires a comparison of all values.

Selecting the best domain decomposition is further complicated due to optimizing single node performance. In this example, it is advantageous to apply BLAS3[2] operations for the local computations. These operations make use of blocks of rows to improve cache utilization. Blocks of rows can only be obtained if a block-cyclic distribution is applied, i.e., columns are not distributed individually but blocks of columns are cyclically distributed.

This discussion makes clear, that choosing a domain decomposition is a very complicated step in program development. It requires deep knowledge of the algorithm's data access patterns as well as the ability to predict the resulting communication.

2 Programming Models

Programming parallel computers is almost always done via the so-called *Single Program Multiple Data* (SPMD) model. SPMD means that the same program (executable code) is executed on all processors taking part in the computation, but it computes on different parts of the data which were distributed over the processors based on a specific domain decomposition. If computations are only allowed on specific processors, this has to be explicitly programmed by using conditional programming constructs (e.g., with `if` or `where` statements). There are two main programming models, *message passing* and *shared memory*, offering different features for implementing applications parallelized by domain decomposition.

The message passing model is based on a set of processes with private data structures. Processes communicate by exchanging messages with special send and receive operations. It is a natural fit for programming distributed memory machines but also can be used on shared memory computers. The domain decomposition is implemented by developing a code describing the local computations and local data structures of a single process. Thus, global arrays have to be split up and only the local part has to be allocated in a process. This handling of global data structures is called *data distribution*. Computations on the global arrays also have to be transformed, e.g., by adapting the loop bounds, to ensure that only local array elements are computed. Access to remote elements has to be implemented via explicit communication, temporary variables have to be allocated, messages have to be constructed and transmitted to the target process.

The shared memory model is based on a set of threads that is created when parallel operations are executed. This type of computation is also called *fork-join parallelism*. Threads share a global address space and thus access array elements via a global index. The main parallel operations are *parallel loops* and *parallel sections*. Parallel loops are executed by a set of threads also called a *team*. The iterations are distributed among the threads according to a predefined strategy. This scheduling strategy implements the chosen domain decomposition. Parallel sections are also executed by a team of threads but the tasks assigned to the threads implement different operations. This feature can for example be applied if domain decomposition itself does not generate enough parallelism and whole operations can be executed in parallel since they access different data structures.

In the shared memory model, the distribution of data structures onto the node memories is not enforced by decomposing global arrays into local arrays, but the global address space is distributed onto the memories by the operating system. For example, the pages of the virtual address space can be distributed cyclically or can be assigned at first touch. The chosen domain decomposition thus has to take into account the granularity of the distribution, i.e., the size of pages, as well as the system-dependent allocation strategy.

While the domain decomposition has to be hard-coded into the message passing program, it can easily be changed in a shared memory program by selecting a different scheduling strategy for parallel loops.

Another advantage of the shared memory model is that automatic and incremental parallelization is supported. While automatic parallelization leads to a first working parallel program, its efficiency typically needs to be improved. The reason for this is that parallelization techniques work on a loop-by-loop basis and do not globally optimize the parallel code via a domain decomposition. In addition, dependence analysis, the prerequisite for automatic parallelization, is limited to access patterns known at compile time. The biggest disadvantage of this model is that it can only be used on shared memory computers.

In the shared memory model, a first parallel version is relatively easy to implement and can be incrementally tuned. In the message passing model instead, the program can be tested only after finishing the full implementation. Subsequent tuning by adapting the domain decomposition is usually time consuming.

3 MPI

The Message Passing Interface (MPI)[8–12] was mainly developed between 1993 and 1997. It is a community standard which standardizes the calling interface for a communication and synchronization function library. It provides Fortran 77, Fortran 90, C and C++ language bindings. It includes routines for point-to-point communication, collective communication, one-sided communication, parallel IO, and dynamic task creation. Currently, almost all available open-source and commercial MPI implementations support the 2.0 standard with the exception of dynamic task creation, which is only implemented by a few. In 2008, an update and clarification of the standard was published as Version 2.1 and work has begun to define further enhancements (version 3.x). For a simple example see the appendix.

3.1 MPI Basic Routines

MPI consists of more than 320 functions. But realistic programs can already be developed based on no more than six functions:

MPI_Init initializes the library. It has to be called at the beginning of a parallel operation before any other MPI routines are executed.
MPI_Finalize frees any resources used by the library and has to be called at the end of the program.
MPI_Comm_size determines the number of processors executing the parallel program.
MPI_Comm_rank returns the unique process identifier.
MPI_Send transfers a message to a target process. This operation is a blocking send operation, i.e., it terminates when the message buffer can be reused either because the message was copied to a system buffer by the library or because the message was delivered to the target process.
MPI_Recv receives a message. This routine terminates if a message was copied into the receive buffer.

3.2 MPI Communicator

All communication routines depend on the concept of a *communicator*. A communicator consists of a process group and a communication context. The processes in the process group are numbered from zero to process count - 1. The process number returned by MPI_Comm_rank is the identification in the process group of the communicator which is passed as a parameter to this routine.

The communication context of the communicator is important in identifying messages. Each message has an integer number called a *tag* which has to match a given selector in the corresponding receive operation. The selector depends on the communicator and thus on the communication context. It selects only messages with a fitting tag and having been sent relative to the same communicator. This feature is very useful in building parallel libraries since messages sent inside the library will not interfere with messages outside if a special communicator is used in the library. The default communicator that includes all processes of the application is MPI_COMM_WORLD.

3.3 MPI Collective Operations

Another important class of operations are *collective operations*. Collective operations are executed by a process group identified via a communicator. All the processes in the group have to perform the same operation. Typical examples for such operations are:

MPI_Barrier synchronizes all processes. None of the processes can proceed beyond the barrier until all the processes started execution of that routine.
MPI_Bcast allows to distribute the same data from one process, the so-called *root* process, to all other processes in the process group.
MPI_Scatter also distributes data from a root process to a whole process group, but each receiving process gets different data.
MPI_Gather collects data from a group of processes at a root process.

MPI_Reduce performs a global operation on the data of each process in the process group. For example, the sum of all values of a distributed array can be computed by first summing up all local values in each process and then summing up the local sums to get a global sum. The latter step can be performed by the reduction operation with the parameter MPI_SUM. The result is delivered to a single target processor.

3.4 MPI IO

Data parallel applications make use of the IO subsystem to read and write big data sets. These data sets result from replicated or distributed arrays. The reasons for IO are to read input data, to pass information to other programs, e.g., for visualization, or to store the state of the computation to be able to restart the computation in case of a system failure or if the computation has to be split into multiple runs due to its resource requirements.

IO can be implemented in three ways:

1. Sequential IO

   A single node is responsible to perform the IO. It gathers information from the other nodes and writes it to disk or reads information from disk and scatters it to the appropriate nodes. Whereas this approach might be feasible for small amounts of data, it bears serious scalability issues, as modern IO subsystems can only be utilized efficiently with parallel data streams and aggregated waiting time increases rapidly at larger scales.

2. Private IO

   Each node accesses its own files. The big advantage of this implementation is that no synchronization among the nodes is required and very high performance can be obtained. The major disadvantage is that the user has to handle a large number of files. For input the original data set has to be split according to the distribution of the data structure and for output the process-specific files have to be merged into a global file for post-processing.

3. Parallel IO

   In this implementation all the processes access the same file. They read and write only those parts of the file with relevant data. The main advantages are that no individual files need to be handled and that reasonable performance can be reached. The parallel IO interface of MPI provides flexible and high-level means to implement applications with parallel IO.

Files accessed via MPI IO routines have to be opened and closed by collective operations. The open routine allows to specify hints to optimize the performance such as whether the application might profit from combining small IO requests from different nodes, what size is recommended for the combined request, and how many nodes should be engaged in merging the requests.

The central concept in accessing the files is the *view*. A view is defined for each process and specifies a sequence of data elements to be ignored and data elements to be read or written by the process. When reading or writing a distributed array the local information can be described easily as such a repeating pattern. The IO operations read and write a number of data elements on the basis of the defined view, i.e., they access the local information only. Since the views are defined via runtime routines prior to the access, the information can be exploited in the library to optimize IO.

MPI IO provides blocking as well as nonblocking operations. In contrast to blocking operations, the nonblocking ones only start IO and terminate immediately. If the program depends on the successful completion of the IO it has to check it via a test function. Besides the collective IO routines which allow to combine individual requests, also non-collective routines are available to access shared files.

3.5 MPI Remote Memory Access

Remote memory access (RMA) operations (also called *one-sided communication*) allow to access the address space of other processes without participation of the other process. The implementation of this concept can either be in hardware, such as in the CRAY T3E, or in software via additional threads waiting for requests. The advantages of these operations are that the protocol overhead is much lower than for normal send and receive operations and that no polling or global communication is required for setting up communication.

In contrast to explicit message passing where synchronization happens implicitly, accesses via RMA operations need to be protected by explicit synchronization operations.

RMA communication in MPI is based on the *window concept*. Each process has to execute a collective routine that defines a window, i.e., the part of its address space that can be accessed by other processes.

The actual access is performed via *put* and *get* operations. The address is defined by the target process number and the displacement relative to the starting address of the window for that process.

MPI also provides special synchronization operations relative to a window. The MPI_Win_fence operation synchronizes all processes that make some address ranges accessible to other processes. It is a collective operation that ensures that all RMA operations started before the fence operation terminate before the target process executes the fence operation and that all RMA operations of a process executed after the fence operation are executed after the target process executed the fence operation. There are also more fine grained synchronization methods available in the form of the General Active Target Synchronization or via locks.

4 OpenMP

OpenMP[13–15] is a directive-based programming interface for the shared memory programming model. It consists of a set of directives and runtime routines for Fortran 77 (published 1997), for Fortran 90 (2000), and a corresponding set of pragmas for C and C++ (1998). In 2005, a combined Fortran, C, and C++ standard (Version 2.5) and 2008, an update (Version 3.0) were published.

Directives are special comments that are interpreted by the compiler. Directives have the advantage that the code is still a sequential code that can be executed on sequential machines (by ignoring the directives/pragmas) and therefore there is no need to maintain separate sequential and parallel versions.

Directives start and terminate parallel regions. When the master thread hits a parallel region a team of threads is created or activated. The threads execute the code in parallel and are synchronized at the beginning and the end of the computation. After the final synchronization the master thread continues sequential execution after the parallel region. The main directives are:

!$OMP PARALLEL DO specifies a loop that can be executed in parallel. The DO loop's iterations can be distributed among the set of threads according to various scheduling strategies including STATIC(CHUNK), DYNAMIC(CHUNK), and GUIDED(CHUNK). STATIC(CHUNK) distribution means that the set of iterations are consecutively distributed among the threads in blocks of CHUNK size (resulting in block and cyclic distributions). DYNAMIC(CHUNK) distribution implies that iterations are distributed in blocks of CHUNK size to threads on a first-come-first-served basis. GUIDED (CHUNK) means that blocks of exponentially decreasing size are assigned on a first-come-first-served basis. The size of the smallest block is determined by CHUNK size.

!$OMP PARALLEL SECTIONS starts a set of sections that are each executed in parallel by a team of threads.

!$OMP PARALLEL introduces a code region that is executed redundantly by the threads. It has to be used very carefully since assignments to global variables will lead to conflicts among the threads and possibly to nondeterministic behavior.

!$OMP DO / FOR is a work sharing construct and may be used within a parallel region. All the threads executing the parallel region have to cooperate in the execution of the parallel loop. There is no implicit synchronization at the beginning of the loop but a synchronization at the end. After the final synchronization all threads continue after the loop in the replicated execution of the program code.
The main advantage of this approach is that the overhead for starting up the threads is eliminated. The team of threads exists during the execution of the parallel region and need not be built before each parallel loop.

!$OMP SECTIONS is also a work sharing construct that allows the current team of threads executing the surrounding parallel region to cooperate in the execution of the parallel sections.

!$OMP TASK is only available with the new version 3.0 of the standard and greatly simplifies the parallelization on non-loop constructs by allowing to dynamically specify portions of the programs which can run independently.

Program data can either be shared or private. While threads do have their own copy of private data, only one copy exists of shared data. This copy can be accessed by all threads. To ensure program correctness, OpenMP provides special synchronization constructs. The main constructs are *barrier synchronization* enforcing that all threads have reached this synchronization operation before execution continues and *critical sections*. Critical sections ensure that only a single thread can enter the section and thus, data accesses in such a section are protected from race conditions. For example, a common situation for a critical section is the accumulation of values. Since an accumulation consists of a read and a write operation unexpected results can occur if both operations are not surrounded by a critical section. For a simple example of an OpenMP parallelization see the appendix.

5 Parallel Debugging

Debugging parallel programs is more difficult than debugging sequential programs not only since multiple processes or threads need to be taken into account but also because program behavior might not be deterministic and might not be reproducible. These problems are not solved by current state-of-the-art commercial parallel debuggers. They only deal with the first problem by providing menus, displays, and commands that allow to inspect individual processes and execute commands on individual or all processes.

Two widely used debuggers are TotalView from Totalview Technologies[21] and DDT from Allinea[3]. They provide breakpoint definition, single stepping, and variable inspection for parallel programs via an interactive interface. The programmer can execute those operations for individual processes and groups of processes. They also provides some means to summarize information such that equal information from multiple processes is combined into a single information and not repeated redundantly. They also support MPI and OpenMP programs on many platforms.

6 Parallel Performance Analysis

Performance analysis is an iterative subtask during program development. The goal is to identify program regions that do not perform well. Performance analysis is structured into three phases:

1. Measurement

   Performance analysis is done based on information on runtime events gathered during program execution. The basic events are, for example, cache misses, termination of a floating point operation, start and stop of a subroutine or message passing operation. The information on individual events can be summarized during program execution (*profiling*) or individual trace records can be collected for each event (*tracing*).

2. Analysis

   During analysis the collected runtime data are inspected to detect *performance problems*. Performance problems are based on *performance properties*, such as the existence of message passing in a program region, which have a condition for identifying it and a severity function that specifies its importance for program performance.

   Current tools support the user in checking the conditions and the severity by a visualization of the program behavior. Future tools might be able to automatically detect performance properties based on a specification of possible properties. During analysis the programmer applies a threshold. Only performance properties whose severity exceeds this threshold are considered to be performance problems.

3. Ranking

   During program analysis the severest performance problems need to be identified. This means that the problems need to be ranked according to the severity. The most severe problem is called the *program bottleneck*. This is the problem the programmer tries to resolve by applying appropriate program transformations.

Current techniques for performance data collection are *profiling* and *tracing*. Profiling collects summary data only. This can be done via *sampling*. The program is regularly interrupted, e.g., every 10 ms, and the information is added up for the source code location which was executed in this moment. For example, the UNIX profiling tool *prof* applies this technique to determine the fraction of the execution time spent in individual subroutines.

A more precise profiling technique is based on *instrumentation*, i.e., special calls to a *monitoring library* are inserted into the program. This can either be done in the source code by the compiler or specialized tools, or can be done in the object code. While the first approach allows to instrument more types of regions, for example, loops and vector statements, the latter allows to measure data for programs where no source code is available. The monitoring library collects the information and adds it to special counters for the specific region.

Tracing is a technique that collects information for each event. This results, for example, in very detailed information for each instance of a subroutine and for each message sent to another process. The information is stored in specialized trace records for each event type. For example, for each start of a send operation, the time stamp, the message size and the target process can be recorded, while for the end of the operation, the time stamp and bandwidth are stored.

The trace records are stored in the memory of each process and are written to a trace file either when the buffer is filled up or when the program terminates. The individual trace files of the processes are merged together into one trace file ordered according to the time stamps of the events.

Profiling has the advantage to be of moderate size while trace information tends to be very large. The disadvantage of profiling is that it is not fine grained; the behavior of individual instances of subroutines can for example not be investigated since all the information has been summed up.

Widely used performance tools include TAU[19,20] from the University of Oregon, Vampir[22,23] from the Technical University of Dresden, and Scalasca[17,18] from the Jülich Supercomputing Centre.

7 Summary

This article gave an overview of parallel programming models as well as programming tools. Parallel programming will always be a challenge for programmers. Higher-level programming models and appropriate programming tools only facilitate the process but do not make it a simple task.

While programming in MPI offers the greatest potential performance, shared memory programming with OpenMP is much more comfortable due to the global style of the resulting program. The sequential control flow among the parallel loops and regions matches much better with the sequential programming model all the programmers are trained for.

Although programming tools were developed over years, the current situation seems not to be very satisfying. Program debugging is done per thread, a technique that does not scale to larger numbers of processors. Performance analysis tools do also suffer scalability limitations and, in addition, the tools are complicated to use. The programmers have to be experts for performance analysis to understand potential performance problems, their proof conditions, and their severity. In addition they have to be experts for powerful but also complex user interfaces.

Future research in this area has to try to automate performance analysis tools, such that frequently occurring performance problems can be identified automatically. First automatic tools are already available: ParaDyn[7] from the University of Wisconsin-Madison and KOJAK[6]/Scalasca[17, 18] from the Jülich Supercomputing Centre.

A second important trend that will effect parallel programming in the future is the move towards clustered shared memory systems with nodes consisting of multi-core processors. This introduces a potentially 3-level parallelism hierarchy (machine - node - processor). Clearly, a hybrid programming approach will be applied on those systems for best performance, combining message passing between the individual SMP nodes and shared memory programming in a node. This programming model will lead to even more complex programs and program development tools have to be enhanced to be able to help the user in developing these codes.

A promising approach to reduce complexity in parallel programming in the future are so-called *partitioned global address space* (PGAS) languages[16], such as Unified Parallel C (UPC) or Co-array Fortran (CAF) which provide simple means to distribute data and communicate implicitly via efficient one-sided communication.

Appendix

This appendix provides three versions of a simple example of a scientific computation. It computes the value of π by numerical integration:

$$\pi = \int_0^1 f(x)dx \quad \text{with} \quad f(x) = \frac{4}{1+x^2}$$

This integral can be approximated numerically by the midpoint rule:

$$\pi \approx \frac{1}{n} \int_1^n f(x_i) \quad \text{with} \quad x_i = \frac{(i-0.5)}{n} \quad \text{for} \quad i = 1, \ldots, n$$

Larger values of the parameter n will give us more accurate approximations of π. This is not, in fact, a very good way to compute π, but it makes a good example because it has the typical, complete structure of a numerical simulation program (initialization - loop-based calculation - wrap-up), and the whole source code fits one one page or slide.

To parallelize the example, each process/thread computes and adds up the areas for a different subset of the rectangles. At the end of the computation, all of the local sums are combined into a global sum representing the value of π.

MPI Version of Example Program

The following listing shows a Fortran90 implementation of the π numerical integration example parallelized with the help of MPI.

```
 1  program pi_mpi
 2  implicit none
 3  include 'mpif.h'
 4  integer          :: i, n, ierr, myrank, numprocs
 5  double precision :: f, x, sum, pi, h, mypi
 6
 7  call MPI_Init(ierr)
 8  call MPI_Comm_rank(MPI_COMM_WORLD, myrank, ierr)
 9  call MPI_Comm_size(MPI_COMM_WORLD, numprocs, ierr)
10
11  if ( myrank == 0 ) then
12     write(*,*) "number of intervals?"
13     read(*,*) n
14  end if
15
16  call MPI_Bcast(n, 1, MPI_INTEGER, 0, MPI_COMM_WORLD, ierr)
17
18  h = 1.0d0 / n
19  sum = 0.0d0
20  do i = myrank+1, n, numprocs
21     x = (i - 0.5d0) * h
22     sum = sum + (4.d0/(1.d0 + x*x))
23  end do
24  mypi = h * sum
25
26  call MPI_Reduce(mypi, pi, 1, MPI_DOUBLE_PRECISION, &
27                  MPI_SUM, 0, MPI_COMM_WORLD, ierr)
28
29  if ( myrank == 0 ) then
30    write(*, fmt="(A, F16.12)") "Value of pi is ", pi
31  endif
32
33  call MPI_Finalize(ierr)
34  end program
```

First, the MPI system has to be initialized (lines 7 to 9) and terminated (line 33) with the necessary MPI calls. Next, the input of parameters (line 11 to 14) and the output of results (lines 29 to 31) has to be restricted so that it is only executed by one processor. Then, the input has to be broadcasted to the other processors (line 16). The biggest (and most complicated) change is to program the distribution of work and data. The do-loop in line 20 has to be changed so that each processor only calculates and summarizes its part of the distributed computations. Finally, the reduce call in lines 26/27 collects the local sums and delivers the global sum to processor 0.

Sequential and OpenMP Version of Example Program

The following listing shows the corresponding implementation of the π integration example using OpenMP. As one can see, because of the need to explicitly program all aspects of the parallelization, the MPI version is almost twice as long as the OpenMP version. Although this is clearly more work, it gives a programmer much more ways to express and control parallelism. Also, the MPI version will run on all kinds of parallel computers, while OpenMP is restricted to the shared memory architecture.

As OpenMP is based on directives (which are plain comments in a non-OpenMP compilation mode), it is at the same time also a sequential implementation of the example.

```
 1  program pi_omp
 2  implicit none
 3  integer            :: i, n
 4  double precision   :: f, x, sum, pi, h
 5
 6  write(*,*) "number␣of␣intervals?"
 7  read(*,*) n
 8
 9  h = 1.0d0 / n
10  sum = 0.0d0
11  !$omp parallel do private(i,x) reduction(+:sum)
12  do i = 1, n
13     x = (i - 0.5d0) * h
14     sum = sum + (4.d0/(1.d0 + x*x))
15  end do
16  pi = h * sum
17
18  write(*, fmt="(A,␣F16.12)") "Value␣of␣pi␣is␣", pi
19  end program
```

The OpenMP directive in line 11 declares the following do-loop as parallel resulting in a concurrent execution of loop iterations. As the variables `i` and `x` are used to store values during the execution of the loop, they have to be declared private, so that each thread executing iterations has its own copy. The variable `h` is only read, so it can be shared. Finally, it is specified that there is a reduction (using addition) over the variable `sum`.

References

1. D. P. Bertsekas and J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall (1989).
2. J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, *ACM Trans. Math. Soft.*, 16:1–17, (1990).
3. Allinea: *DDT*, `http://allinea.com/`.
4. I. Foster, *Designing and Building Parallel Programs*, Addison Wesley (1994).

5. G. Fox, *Domain Decomposition in Distributed and Shared Memory Environments*, International Conference on Supercomputing June 8-12, 1987, Athens, Greece, Lecture Notes in Computer Science 297, edited by C. Polychronopoulos (1987).
6. F. Wolf and B. Mohr, Automatic Performance Analysis of Hybrid MPI/OpenMP Applications. *Journal of Systems Architecture*, 49(10–11):421–439 (2003).
7. B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvine, K. L. Karavanic, K. Kunchithapadam, and T. Newhall, The Paradyn Parallel Performance Measurement Tool, *IEEE Computer*, Vol. 28, No. 11, 37–46 (1995).
8. MPI Forum: *Message Passing Interface*, `http://www.mpi-forum.org/`.
9. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI - the Complete Reference, Volume 1, The MPI Core,* 2nd ed., MIT Press (1998).
10. W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir, *MPI - the Complete Reference, Volume 2, The MPI Extensions*, MIT Press (1998).
11. W. Gropp, E. Lusk, A. , Skjellum, *Using MPI, 2nd Edition*, MIT Press (1999).
12. W. Gropp, E. Lusk, R. Thakur, *Using MPI-2: Advanced Features of the Message Passing Interface*, MIT Press (1999).
13. OpenMP Forum: *OpenMP Standard*, `http://www.openmp.org/`.
14. L. Dagum and R. Menon, *OpenMP: An Industry-Standard API for Shared-memory Programming*, IEEE Computational Science & Engineering, Vol. 5, No. 1, 46–55 (1998).
15. B. Chapman, G. Jost, R. van der Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*, MIT Press (2007).
16. C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanty, Y. Yao, An Evaluation of Global Address Space Languages: Co-Array Fortran and Unified Parallel C, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 2005)*, ACM, (2005)
17. M. Geimer, F. Wolf, B. J. N. Wylie, B. Mohr, Scalable parallel trace-based performance analysis. *Proc. 13th European PVM/MPI Users' Group Meeting*, Bonn, Germany. Volume 4192 of LNCS, Springer, pp. 303–312, (2006).
18. B. J. N. Wylie, M. Geimer, F. Wolf, Performance measurement and analysis of large-scale parallel applications on leadership computing systems, *Scientific Programming*, 16(2–3), Special Issue on Large-Scale Programming Tools and Environments, pp. 167–181, (2008).
19. A. Malony, S. Shende, A. Morris, Performance technology for productive parallel computing, *Advanced Scientific Computing Research Computer Science FY2005 Accomplishments*, U.S. Dept. of Energy Office of Science, (2005).
20. S. Shende, A. Malony, The TAU parallel performance system, *International Journal of High Performance Computing Applications*, 20, pp. 287–331, SAGE Publications, (2006).
21. Totalview Technologies: *TotalView*, `http://www.totalviewtech.com/`.
22. H. Brunst, W. E. Nagel, Scalable performance analysis of parallel systems: Concepts and experiences. *Proc. of the Parallel Computing Conference (ParCo)*, Dresden, Germany, pp. 737–744, (2003).
23. H. Brunst, *Integrative Concepts for Scalable Distributed Performance Analysis and Visualization of Parallel Programs*, PhD thesis, TU Dresden, Shaker, Aachen, (2008).