John von Neumann Institute for Computing

Parallel Programming Models, Tools and Performance Analysis

Bernd Mohr, Michael Gerndt

published in

Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms, Lecture Notes,
J. Grotendorst, D. Marx, A. Muramatsu (Eds.),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **10**, ISBN 3-00-009057-6, pp. 507-520, 2002.

http://www.fz-juelich.de/nic-series/volume10

Parallel Programming Models, Tools and Performance Analysis

Bernd Mohr[1] **and Michael Gerndt**[2]

[1] John von Neumann Institute for Computing
Central Institute for Applied Mathematics
Research Centre Jülich, 52425 Jülich, Germany
E-mail: b.mohr@fz-juelich.de

[2] Technische Universität München
Institut für Informatik, LRR
80290 München, Germany
E-mail: gerndt@in.tum.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

Although the performance of sequential computers increases incredibly fast, it is insufficient for a large number of challenging applications. Applications requiring much more performance are numerical simulations in industry and research as well as commercial applications such as query processing, data mining, and multi-media applications. Architectures offering high performance do not only exploit parallelism on a very fine grain within a single processor but apply a medium to large number of processors concurrently to a single application. High-end parallel computers deliver up to 30 Teraflop/s (10^{12} floating point operations per second) and are developed and exploited within the ASCI (Accelerated Strategic Computing Initiative) program of the Department of Energy in the USA.

This article concentrates on programming numerical applications on distributed memory computers introduced in Sec. 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 (Sec. 1.2)[7].

The parallel implementation is then based on either the message passing or the shared memory model (Sec. 2). The standard programming interface for the message passing model is MPI (Message Passing Interface)[14,11], offering a complete set of communication routines (Sec. 2.1). OpenMP[4,13] is the standard for directive-based shared memory programming and will be introduced in Sec. 2.2.

Since parallel programs exploit multiple threads of control, debugging is even more complicated than for sequential programs. Sec. 3 outlines the main concepts of parallel debuggers and presents TotalView[15], the most widely available debugger 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 algorithm 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. Sec. 4 presents basic performance analysis techniques and introduces the widely available performance analysis tools VAMPIR[16] (for MPI programs) and GuideView[9] (for OpenMP).

1.1 Parallel Architectures

Parallel computers that scale beyond a small number of processors circumvent the main memory bottleneck by distributing the memory among the processors. Current architectures[3] are composed of single-processor nodes with local memory or of multiprocessor nodes where each node's main memory is shared among its processors. The latter are often called SMP (Symmetrical Multi Processor) nodes.

The most important characteristic of this *distributed memory 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.

Three 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. Processors obtain data from remote memory only by exchanging messages between processes on the requesting and the supplying node.

Remote Memory Access (RMA) computers allow to access remote memory via specialized operations implemented by hardware. The accessed 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.

Cache-Coherent Non Uniform Memory Access (ccNUMA) computers do have a shared physical address space. 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. Coherence algorithms ensure that multiple copies of a cache line are kept coherent, i.e., the copies do have the same value.

While most of the early parallel computers were NORMA systems, today's systems are either RMA or ccNUMA computers. This is because remote memory access is a lightweight communication protocol that is more efficient than standard message passing since data copying and process synchronization are eliminated. In addition, 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.

Typical examples of the three classes are clusters of workstations (NORMA), CRAY T3E (RMA), and SGI Origin 3000 (ccNUMA).

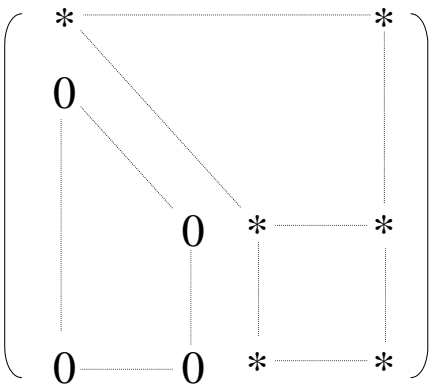

Figure 1. Structure of the matrix during Gaussian elimination.

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 according to a selected *domain decomposition*[8]. Processors execute those operations that determine new values for local elements (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[2]. The main structure of the matrix during the iterations of the algorithm is outlined in Fig. 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 algorithms 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 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 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

The two main programming models, *message passing* and *shared memory*, offer 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. 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 have 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 are 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.

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.

2.1 MPI

The Message Passing Interface (MPI)[14, 11] was developed between 1993 and 1997. It includes routines for point-to-point communication, collective communication, one-sided communication, and parallel IO. While the basic communication primitives have already been defined since 1994 and implemented on almost all parallel computers, remote memory access and parallel IO routines are part of MPI 2.0 and are only available on few machines.

2.1.1 MPI Basic Routines

MPI consists of more than 120 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.

2.1.2 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.

2.1.3 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.

2.1.4 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. While the IO is sequential and thus need not be parallelized, the full performance of the IO subsystem might not be utilized. Modern systems provide high performance IO subsystems that are fast enough to support multiple IO requests from different nodes in parallel.

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 splitted according to the distribution of the data structure and for output the process-specific files have to be merged into a global file for postprocessing.

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 disadvantage is that it is difficult to reach the same performance as with private IO. 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.

2.1.5 MPI Remote Memory Access

Remote memory access (RMA) operations (also called *1-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.

2.2 OpenMP

OpenMP[4, 13] 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).

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 and thus no two versions, a sequential and a parallel version, need to be maintained.

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 in various ways including STATIC(CHUNK), DYNAMIC(CHUNK), and GUIDED(CHUNK) among the set of threads. 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 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 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.

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.

3 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 deal only with the first problem by providing menus, displays, and commands that allow to inspect individual processes and execute commands on individual or all processes.

The widely used debugger is TotalView from Etnus Inc[15]. It provides breakpoint definition, single stepping, and variable inspection via an interactive interface. The programmer can execute those operations for individual processes and groups of processes. TotalView also provides some means to summarize information such that equal information from multiple processes is combined into a single information and not repeated redundantly. It also supports MPI and OpenMP programs on many platforms.

4 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 four 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 or individual trace records can be collected for each event.

   Summary information has the advantage to be of moderate size while trace information tends to be very large. The disadvantage 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.

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 visualizing 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.

4. Refinement

   The performance problems detected in the previous phases might not be precise enough to allow the user to start optimization. At the beginning of performance analysis, summary data can be used to identify critical regions. The summary data might not be sufficient to identify why, for example, a region has high message passing overhead. The reason, e.g., very big messages or load imbalance, can be identified only with more detailed information. Therefore the performance problem should be refined into hypotheses about the real reason and additional information be collected in the next performance analysis cycle.

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.

The following sections describe two widely available performance analysis tools for MPI programs (VAMPIR) and OpenMP applications (GuideView).

4.1 VAMPIR

VAMPIR (Visualization and Analysis of MPI Resources) is an event trace analysis tool[12, 16] initially developed by the Central Institute for Applied Mathematics of the Research Centre Jülich and now is commercially distributed by the German company PALLAS. VAMPIR has three components:

- The VAMPIR tool itself is a graphical event trace browser implemented for the X11 Window system using the Motif toolkit. It is available for all major UNIX platforms.

- The VAMPIR runtime library (VampirTrace) provides an API for collecting, buffering, and generating event traces as well as a set of wrapper routines for MPI and shmem communication routines which record message traffic in the event trace.

- In order to observe functions or subroutines in the user program, their entry and exit has to be instrumented by inserting calls to the VAMPIR runtime library. Observing message passing functions is handled by linking the program with the VAMPIR wrapper function library.

  VAMPIR comes with a source instrumenter for ANSI Fortran 77. Programs written in other programming languages (e.g., C or C++) have to be instrumented manually.

During the execution of the instrumented user program, the VAMPIR runtime library records entry and exits to instrumented user and message passing functions and the sending and receiving of messages. For each message, its tag, communicator, and length is recorded. Through the use of a configuration file, it is possible to switch the runtime observation of specific functions on and off. This way, the program doesn't have to be re-instrumented and re-compiled for every change in the instrumentation.

Large parallel programs consist of dozens or even hundreds of functions. To ease the analysis of such complex programs, VAMPIR arranges the functions into groups, e.g., user functions, MPI routines, I/O routines, and so on. The user can control/change the assignment of functions to groups and can also define new groups.

VAMPIR provides a wide variety of graphical displays to analyze the recorded event traces:

- The dynamic behavior of the program can be analyzed by timeline diagrams for either the whole program or a selected set of nodes. By default, the displays show the whole event trace, but the user can zoom-in to any arbitrary region of the trace. Also, the user can change the display style of the lines representing messages based on their tag/communicator or the length. This way, message traffic of different modules or libraries can easily be visually separated.

- The parallelism display shows the number of nodes in each function group over time. This allows to easily locate specific parts of the program, e.g., parts with heavy message traffic or IO.
- VAMPIR also provides a large number of statistical displays. It calculates how often each function or group of functions was called and the time spent in them. Message statistics show the number of messages sent, and the minimum, maximum, sum, and average length or transfer rate between any two nodes. The statistics can be displayed as barcharts, histograms, or textual tables.

  A very useful feature of VAMPIR is that the statistic displays can be linked to the timeline diagrams. By this, statistics can be calculated for any arbitrary, user selectable part of the program execution.
- If the instrumenter/runtime library provides the necessary information in the event trace header, the information provided by VAMPIR can be related back to the source code. VAMPIR provides a source code and a call graph display to show selected functions or the location of the send and the receive of a selected message.

In summary, VAMPIR is a very powerful and highly configurable event trace browser. It displays trace files in a variety of graphical views, and provides flexible filter and statistical operations that condense the displayed information to a manageable amount. Rapid zooming and instantaneous redraw allow to identify and focus on the time interval of interest.

4.2 GuideView

GuideView[9] is the integrated profiling performance analysis component of the OpenMP Compilation Environment KAP/Pro of KAI. It can be used to look for typical OpenMP performance problems like load imbalance, false sharing, or excessive synchronization.

The necessary instrumentation for performance data collection is automatically inserted on user request by the Guide OpenMP compiler. During program execution, the Guide runtime system collects execution statistics for each OpenMP construct in each thread. Execution time is measured and categorized into user code execution in sequential and parallel mode, sequential and parallel overhead, time spent in lock functions and barriers, as well as load imbalance at barriers. Afterwards, the collected performance data can be analyzed with the GuideView tool. It provides performance visualizations for the whole program, on a per thread basis, and on a per OpenMP region basis.

In addition, performance data files from different program runs can be loaded and analyzed simultaneously. This allows to compare the program performance based on different input datasets and/or thread numbers.

5 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 program 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. It is the goal of the IST working group APART on *Automatic Performance Analysis: Resources and Tools* to investigate base technologies for future more intelligent tools[1]. An important result of this work is a collection of performance problems for parallel programs that have been formalized with the ASL, the *APART Specification Language*[6]. This approach will lead to a formal representation of the knowledge applied in the manually executed performance analysis process and thus will make this knowledge accessible for automatic processing. First automatic tools are already available: ParaDyn[10] from the University of Wisconsin-Madison, Kappa-PI[5] from the Universitat Autonoma de Barcelona, and EXPERT[17,18] from the Research Centre Jülich.

A second important trend that will effect parallel programming in the future is the move towards clustered shared memory systems. 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.

References

1. APART: *IST Working Group on Automatic Performance Analysis Resources and Tools*, `http://www.fz-juelich.de/apart/`, 2001.
2. D. P. Bertsekas, J. N. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*, Prentice-Hall, ISBN 0-13-648759-9, 1989.
3. D. E. Culler, J. P. Singh, A. Gupta, *Parallel Computer Architecture - A Hardware/Software Approach*, Morgan Kaufmann Publishers, ISBN 1-55860-343-3, 1999.
4. L. Dagum, R. Menon, *OpenMP: An Industry-Standard API for Shared-memory Programming*, IEEE Computational Science & Engineering, Vol. 5, No. 1, 46–55, 1998.
5. A. Espinosa, *Automatic Performance Analysis of Parallel Programs*, PhD thesis, Universitat Autonoma de Barcelona, 2000.
6. Th. Fahringer, M. Gerndt, B. Mohr, F. Wolf, G. Riley, J. Träff, *Knowledge Specification for Automatic Performance Analysis*, APART Technical Report, Research Centre Juelich FZJ-ZAM-IB-2001-08, 2001.
7. I. Foster, *Designing and Building Parallel Programs*, Addison Wesley, ISBN 0-201-57594-9, 1994.

8. 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.
9. KAI: *GuideView*, `http://www.kai.com/parallel/openmp.html`, 2001.
10. 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.
11. MPI Forum: *Message Passing Interface*, `http://www.mpi-forum.org`, 2001.
12. W. E. Nagel, A. Arnold, M. Weber, H.C. Hoppe, K. Solchenbach, *VAMPIR: Visualization and Analysis of MPI Resources*, Supercomputer 63, Vol. 12, No. 1, 69–80, 1996.
13. OpenMP Forum: *OpenMP Standard*, `http://www.openmp.org`, 2001.
14. M. Snir, St. Otto, St. Huss-Lederman, D. Walker, J. Dongarra, *MPI - The Complete Reference*, MIT Press, ISBN 0-262-69216-3, 1998.
15. Etnus Inc.: *Totalview*, `http://www.etnus.com/Products/Totalview/`, 2001.
16. Pallas GmbH: *VAMPIR*, `http://www.pallas.de/pages/vampir.htm`, 2001.
17. F. Wolf, B. Mohr, Automatic Performance Analysis of MPI Applications Based on Event Traces, In *Proc. of the European Conference on Parallel Computing (Euro-Par)*, 123–132, Munich (Germany), 2000.
18. F. Wolf, B. Mohr, *Automatic Performance Analysis of SMP Cluster Applications*, Technical Report, Research Centre Juelich FZJ-ZAM-IB-2001-05, 2001.