Problem-oriented approach to MPI applications
development cycle

Sergii Stirenko, Oleksandr Zinenko, Dmitri Gribenko,

High-performance computing center of NTUU “KPI”
Department of computer engineering, NTUU “KPI”

[stirenko,zinenko,gribenko]@hpcc.kpi.ua

Abstract. Applications, highly-loaded in terms of computation, also tend to take significant time during the whole
development cycle, namely while debugging on a real-scale system. Profiling techniques aiming to increase software
performance are time-consuming themselves, therefore it is important to propose an approach decreasing the total devel-
opment time for MPI applications and facilitating where possible the work of programmer. We propose it as a combination
of framework for stating tasks in terms of problems rather than parallel computing primitives with an analysis tool to tune
MPI applications. Used jointly they allow to take control over abstraction levels in the development process.

Keywords

MPI, OpenVampir, OpenMP, granularity, distributed computing, framework, high performance, scientific computing,
programming tool, developer tool.

Introduction

High-performance and high-throughput computing are dramatically evolving branches of modern information technology
that serve principally for solving challenging scientific problems in a wide variety of fundamental and applied sciences.
Numerous propositions and enhancements made for these technologies, multiple research and development projects are
required by research institutions foremost. Existing supercomputing technologies available for general use in scientific
community are mostly loaded with computationally-heavy tasks that take significant time to obtain the results. While
computer performance is frequently limited by technical, financial and even administrative issues [1], the natural way to
obtain results faster relies on speeding up the program itself.

Generally, each program can be exposed to optimization process that might increase its performance. This process,
nevertheless, differ significantly between computer systems the program could be run on. The resulting solution time is
subject to changes due to different processor types, interconnect technologies, operating system and libraries versions, etc.
Therefore, satisfactory optimization would require deep knowledge in computer science which is not necessarily a feature
of any scientist [2].

On the other hand, developing quality scientific applications that are able to show really high performance on modern
hardware is hard and time-consuming task that is usually excluded from the total time spent. A typical user of super-
computer tends to specify his task in terms of his domain rather than using computer-friendly techniques. Even if the
user could perform optimization routines, it would consumed a considerable amount of time during which he could have
advanced in his proper research.

Under that light, developing the approach that could enable a user to describe tasks in a natural problem-oriented way
in his domain is a task of prime importance. Furthermore, facilitating the development process for a common user on all
the stages including debugging and optimizing as well as eliminating the necessity in involving computer specialists in
this process looks very perspective for modern scientific computing.

1 Parallel programming technologies

Over the past decades parallel and distributing programming was developing apparently until it became the ultimate ten-
dency for computational-heavy applications. [3, 4] Apparently, numerous solutions was developed to enable this approach,
but quite a few of them had a remarkable impact on the programming techniques themselves. Let us examine their most
innovative features keeping in mind MPI as a primary technology for scientific distributed computing [5, 6].
The OpenMP technology that includes the set of compiler directives along with the run-time library was proposed for enabling the fast parallelization of existing (and even no longer supported) program code in C and Fortran languages [7]. It is oriented primarily to easily parallelizing loops as the program parts taking the most computational time. The main effect of this technology was the demonstration of the possibility to include a just-in-time scheduler into the runtime library and relieve the user from having to explicitly deal with threading primitives. OpenMP remains a very perspective technology, still it works only on the shared memory systems that considerably limits its use in HPC. The Intel distributed OpenMP project is no longer elaborated, therefore OpenMP finds its usage in combination with traditional MPI.

Intel Threading Building Blocks project (shortly TBB) brings into parallel computing the idea of describing tasks in terms of ranges and their transformations: the initial task is stated as a range and a set of operations to perform on. [8, 9] This allows for dividing the range into several parts and transforming them independently, in other words — in parallel. This technology evolved with time into a sophisticated C++ framework for describing data-parallel and even general tasks in terms of ranges. TBB also uses its own scheduler in the runtime and as well as OpenMP is designed to work in shared memory.

Google’s Map/Reduce framework is usually classified as a big data processing tool or somewhere closer to NoSQL databases, but by itself it tends to be a distributed programming framework. It assumes the task to be specified in terms of “map” operation that performs independent computation and “reduce” operation that combines the computation results, while the data transmission is taken care of by the runtime library. Although it provides a distributed computation engine, it doesn’t use MPI as a backend that renders it barely useful on the modern MPI-enabled computational clusters.

We can delineate a niche for a computational framework that combines the generalized approach of TBB for defining tasks without explicit usage of parallel programming primitives with MPI-based backend for working on most of existing distributed computer system used for scientific applications.

2 Task splitting approach

The proposed approach relies on the structure of a problem to be solved. It supposes the problem to be divisible into subproblems of a lesser size with allowance of recursivity for this operation. It presumes as well that subproblems can be solved by generalized algorithm (the same one is used for all subproblems without regard to size) and their results can be combined to obtain the global result. Although the approach makes no assumptions on the cost of split and merge operation, it is preferable for their total time to be of same or lesser order as main computing algorithm in terms of input data size in \(O\)-notation in order to reach acceptable solution time on a parallel system.

Let us consider a task \(T_0\) that can be trivially decomposed into a set of action sequences \(T_0 = \{t_{0,i}\}\). These sequences \(t_i = \langle a_0, a_1, a_2, \ldots \rangle\) comprise actions that should be performed consequently but can be performed independently between different sequences. Such a decomposition \(T_0 \Rightarrow \{\langle a_{0,i}\rangle\}\) given the minimum total size of all sequences \(t_{0,i}\), namely \(\sum_i \text{size}_{a,i} \rightarrow \min\), describes a theoretically ideal task decomposition for parallel computation. It doesn’t guarantee the least solution time in a real system though due to the presence of another significant implementation factors like caches, superscalar architecture features, data transmission overhead, etc.

Another principal problem concerning the decomposition proposed above is variable granularity size that could result in performance reduction for several computer systems. Therefore, a kind of recursive decomposition would perform better in this case. Let us denote a \(j\)-th subtask \(T_{i}^{(j)}\) of the task \(T_0\), or the \(j\)-th first-order subtask. For the sake of generality the \(T_0\) task can be also treated as the single zero-order subtask. Over that subtasks we can define an operator split \(T_i \rightarrow \{T_{i(a+1)}\}\) that decomposes the subtask of \(i\)-th order into a set of independent \((i+1)\)-th order subtasks so that their sizes tend to be equal (1).

\[
\begin{align*}
T_i^{(a)} &= \bigcup_k T_{i(a+1)}^{(k)}; \\
T_{i(a+1)}^{(a)} \cap T_{i(a+1)}^{(b)} &= 0 \quad \forall a, b.
\end{align*}
\]

This operator can be applied recursively to the subtasks in order to get the required number of subtasks, i.e. multiple of processors number, required size of subtasks even if same-order subtasks have different sizes, decompose the subtasks on the fly if needed for resource-sharing system. It therefore allows for dynamic granularity control as a just-in-time function. The granularity can vary from the initial task size to the minimum subtask size. The later enables us to impose an artificial lower limit for the granularity by defining split to be the identity operator for tasks of a certain order.

The task-splitting approach implies the existence of structured solution algorithm that can be applied to the subtask of any order. The application is denoted by the compute \(T_i^{(a)} \rightarrow R_i^{(a)}\) operator where \(R_i\) stands for \(i\)-th order subresult. This operator in combination with split limitation can cover a wide variety of computational algorithms.

In order to obtain the required result \(R_0\) of the global task \(T_0\) the subresults should be combined according to a certain algorithm. Let us denote it as merge \(\{R_{i(a+1)}^{(a)}\} \rightarrow R_i^{(a)}\) operator. This operator should be only applied to the set of results obtained by computing appropriately the set of corresponding subtasks. In order to perform results combination
correctly we have to introduce another concept named splitting path which represents the sequence of subtask numbers from which the current one was received by division, namely the zero-order subtask has an empty splitting path $\langle \rangle$, the splitting path for a $i$-th first-order subtask contains only $i$: $\langle i \rangle$, the $j$-th second-order subtask obtained from the $i$-th first-order has it equal to $(i, j)$. Splitting path allows for determining the subtask level as well as for combining the corresponding subresults. No different tasks nor results can have the same splitting paths. Subresults can be merged if and only if they splitting paths have equal sizes and differ only by the last term.

The generalized solution algorithm for the tasks that can be described as above is as follows. The initial task $T_0$ is decomposed into subtasks $\{T_x\}$ of possibly different order using split operator. The subtasks are transformed into subresults by applying the compute operator. Then the subresults are combined appropriately to get the result. The stopping criteria for subtask decomposition and task composition are as follows

$$\text{split } T^{(i...k)}_i = \{T^{(i+1)}_{(i+1)}\}, \quad T^{(i...k)}_i = T^{(i+1)}_{(i+1)}; \quad (2)$$

$$\text{merge } \{R_x\} = R_0; \quad (3)$$

Specifically the subtask decomposition stops as soon as it generates set with one element (2) while the composition stops when the global result is computed (3). The later informs about the computation finish as well.

### 3 Task statement isomorphism

The approach to describe a task mentioned in the previous section allows the user to define his problem by the following tuple:

$$\langle ::T, ::R, \text{split, compute, merge, } T_0 \rangle, \quad (4)$$

where

- $::T$ and $::R$ are task and result types respectively (for a type-enabled calculus, i.e. most programming languages);
- $\text{split} :: T \rightarrow \{T\}$ is the subtask decomposition operator;
- $\text{compute} :: T \rightarrow R$ is the main computation algorithm structured so that is can solve a subtask of any order;
- $\text{merge} :: \{R\} \rightarrow R$ is the subresult composition operator;
- $T_0$ is the initial task statement in terms of the type specified above.

The simplest example for such description is functor transformation, also known in programming as mapping over a functor, e.g. adding a constant to each element of a tensor. This example task belongs to the large domain of data-parallel tasks, which can be solved extremely effectively by distributed systems. For this example, both $T$ and $R$ types are tensors; split operation performs a cut of a tensor in one dimension while merge does the opposite; and compute performs requested operation in cycle for each tensor element. The proposed split/merge combination can be used in large number of data-parallel tasks and no more than linear complexity in terms of input data which renders our parallelization approach very efficient.

The isomorphism of classical thread or process-based task statement approach to the proposed one can be shown easily considering the following transformation. Let us suppose a set of $p$ (MPI) processes $\{P_i\}$ that comprise all required computations. Then we can define the initial task to be an array of $p$ integers from in range $[0, p)$. Therefore task type will be represented as an array of integers. The split operation performs decomposition of the array into a set of disjoint subarrays. The compute operator initiates the $P_i$ process for each number $i$ in its input. Thus, regardless of the subtask size, all the initial processes will be run once and perform their instructions. After that, reduction is not necessary, so for the simplicity we can assign its type to be an array of integers as well.

### 4 Approach to development

Classical approach to parallel program development relies on the knowledge of low-level parallel computation primitives such as threads or shared data. This forces user to focus on the implementation details rather than on the problem being programmed. These primitives can differ between available systems: the bigger one is the distinction between shared-memory and distributed computer systems, which requires a completely different approaches to communication between parallel processes. Being the most performant nowadays, the latter implies usage of MPI interface with a slightly complicated procedure-oriented structure.
The proposed approach allows for stating the problem in the programming language without necessity to explicitly interact with parallel and distributed programming mechanisms. Moreover, it enables the user to describe the problem in more abstract terms like task decomposition and result composition that seems more natural for most scientific-related computations: these operations are not closely related to the programming language concepts and restrictions.

Task-splitting approach is subject to criticism due to requirement for a well-structured algorithm for solution, but most typical algorithm do satisfy this constraint. As for complicated computational algorithms — they are not easily programmed in a traditional manner either.

5 Approach to debugging and optimizing

Debugging and optimizing can be both performed by means of our recently developed profiling and optimization tool [10]. It is based on the MPI instrumentation method in modern compilers [11] and MPI implementations [12] in order to circumvent the batch processing mode in common distributed systems as well as to allow the subsequent trace analysis on the system different from the one the program has been executed on. By virtue of the advanced code instrumentation techniques this also allows for tracing and logging all the events from both MPI and user code, separating as system I/O calls and networking operations in the dedicated class [13]. We found preferable to use Open Trace Format in our solution since it is included in the default configuration of many MPI implementations.

Our tool includes a set of useful visual analytical and statistical widgets that enable user to focus on the particularities of parallel computer system if needed leaving uncovered the profiling and debugging techniques already implemented in commonly used tools. Along with this set it includes a special mode for analyzing tasks described in the framework proposed in the previous sections. The latter mode is used for estimating the quality of parallel solution in terms of its time and processor and memory efficiency.

The crucial problem in debugging parallel applications is avoiding locks and infinite loops during the execution. Such kind of issues can be only be discovered while working on the real-scale system. Our approach allows to record the program locked state even if it was killed by the scheduler due to timeout or crashed after which the trace file can be examined independently and could indicate the possible problem.

The tool includes the following major analysis modes. **Timeline** allows a user to analyze visually the execution of all the processes of an application during the specified time period varying from whole duration of program operation to minimal recorded time resolution. This tool also distinguishes different call types enabling the user to see possible bottlenecks or program idle periods caused by waiting for data. **Timeline** has to expose both MPI data point-to-point and collective transfers. Using this kind of analysis and a data dependency graph one might change a program execution flow to ensure computations and data transfer are performed simultaneously wherever possible. In case of abnormal program termination this tool in combination with **process call timeline** can show the function the program was idling in that in turn indicates the possible place of loop or lock. In case of locks, event detailization can show exactly the line of code and the MPI call that caused the lock.

**Heatmap** generally shows the intensity of communication between different processes in terms of average transfer speed and data volumes, again for any user-specified time span. It could be used to deal with architecture of a particular system, e.g. for producing recommendations to place bandwidth-hungry processes as close as possible in a system. It might show whether the program was split into parallel parts optimally with regards to computation to communication ratio or whether it would have been better to merge some strongly connected on a data graph parts. This statistics can be used to hint runtime granularity adjustment system to choose another subtask size for the current or next run.

Different **graph** representations like call tree and process communication graphs allow user to be aware of communication between different processes and to introduce, for example, a communicator in MPI program or create hints for node affinity in a system.

Various **summaries** exposing the most general information about execution process like memory usage or execution time part for data I/O, computations and systems calls as well as user-defined arbitrary counters are assumed to be used for estimating overall program efficiency and possible benefits from parallelizing and running on a distributed system.

All these options can be used as together with aforementioned framework for describing tasks, as separately for debugging, analysis and optimization of traditional MPI programs.

6 Discussion

These approaches to program development and subsequent maintenance were generalized as a single problem-oriented approach to the development cycle since they allow to reduce the time of this cycle significantly by reducing the time required for each step in this cycle independently. Although program development and debugging time is rarely included
in the overall estimate of scientific result obtaining delay, they should be considered at least in cases where there is no possibility to decrease solution time due to objective limitations.

Despite the fact that these approaches can be used separately: an application building framework for stating the task in terms of mathematical problem description and a special tool for program analysis and performance tuning — they are united by the common objective of controlling complexity from the user’s viewpoint; which is treated by some authors as the key concept in programming. They are involved in each other in a more subtle manner, specifically the former can use the results produced by the latter for fine-tuning its decisions and accumulating data for determining consequently the suitable parallelization patterns and granularity hinting for tasks grouped by certain criterion; while the latter can user the results of the former for reasoning about “suitability” of different systems for a particular tasks as well as allowing user to reach the required depth of performance analysis even though his program was written in an abstract domain-oriented manner.

The analysis tool has also shown substantial impact on the educational process as much as it allowed to explicitly demonstrate the real-scale execution of a parallel program on big distributed system along with enough intrinsics of its components. We also used it as an example of abstraction level managing in computer science.

Conclusions

Summing up, we emphasized that the total time between problem statement and final result obtaining in the scientific computing should normally include the time required for developing the software solution if it had not existed previously. This time might exceed the software runtime in several orders of magnitude. Therefore, the time required for this process should be decreased by means of introducing simpler techniques for describing computational tasks to be solved in parallel. Our global aim was to facilitate the work of a scientist — distributed computer system user — by allowing him to state problems in a more natural way along with the possibility to still have enough control over the computational process. Keeping this in mind we proposed a programming framework for describing tasks in terms of (de)compositions and transformations. It performs the computations in parallel relieving the user from the explicit description of parallel computing and allows to use different scheduling approach. The latter, in turn, enables this system to potentially vary the program granularity on the fly or adapt it to a certain computer system. We also enhanced our previously announced tool for dynamic analysis to allow for fine-tuning of programs written in the former framework.

These approach has substantial perspectives in scientific computing. One of the anticipated further development directions is a more sophisticated interaction between runtime part of the framework and the analysis tool in order to make smart distributed granularity-aware scheduling algorithms using the previously collected data classified by distributed system types and problem classes. This might allow further to recognize parallelization patterns in different tasks and automatically produce granularity hints and suitability estimate for a random combination of task and system. This statistical approach to finding suboptimal granularity values could be also enhanced by applying evolutionary and ontological methods as well.

References

[9] Intel thread building blocks documentation [Electronic resource].