Prototyping Framework for Parallel Numerical Computations

Ondřej Meca, Stanislav Böhm(✉), Marek Běhálek, and Martin Šurkovský

Department of Computer Science, FEI VŠB Technical University of Ostrava, Ostrava, Czech Republic
{ondrej.meca,stanislav.bohm,marek.behalek,martin.surkovsky}@vsb.cz

Abstract. Our research is focused on the simplification of parallel programming for distributed memory systems. Our goal is to build a unifying framework for creating, debugging, profiling, and verifying parallel applications. The result of this effort is an open source tool Kaira. In this paper, we focus on prototyping of parallel applications. We have extended Kaira by the ability to generate parallel libraries. More precisely, we present a framework for fast prototyping of parallel numerical computations. We demonstrate our idea on a combination of parallel libraries generated by our tool Kaira and GNU Octave. Hence, a user can verify the idea in a short time, create a real running program and verify its performance and scalability.

Keywords: Prototyping · Parallel computing · Visual programming · Libraries

1 Introduction

Parallel computers are more and more available nowadays. A lot of people participate in developing parallel programs, but there are well-known difficulties of parallel programming. For example the user must learn how to use different specialized tools for profiling or debugging. Also, it usually takes more time to get a working parallel application that can be tested. Therefore, it can be difficult for many non-experts (even if they are experienced programmers of sequential applications) to make their programs run in parallel on a computer cluster.

The overall goal of our research is to reduce some complexity in parallel programming. In this paper, we focus on parallel application prototyping. More precisely, we present a framework for fast prototyping of parallel numerical computations. Hence, a user can verify his/her idea in a short time, create a real running program, and verify its performance and scalability. To address numerical computations, we demonstrate our idea on the combination of parallel libraries generated by our tool Kaira and GNU Octave[1] (Which we will now simply refer to as "Octave" in text). However, this approach can be easily generalized and

[1] http://www.gnu.org/software/octave/

R. Wyrzykowski et al. (Eds.): PPAM 2013, Part II, LNCS 8385, pp. 95–104, 2014.
DOI: 10.1007/978-3-642-55195-6_9, © Springer-Verlag Berlin Heidelberg 2014

generated libraries can be combined with other tools. We have chosen Octave because it represents a good example of a prototyping software, where users can easily experiment with their ideas.

2 Related Work

This section presents tools which have implemented ideas similar to Kaira and from which we have drawn inspiration. First, there were tools for visual programming of parallel applications. These tools were developed mainly in the 90s. As an example we can name *GRADE* [9], *CODE* [11] or HeNCE [3]. It is hard to evaluate these tools because they are no longer available or they run on no longer available hardware or operating systems. The semantics of our tools is similar to CODE but as far we know, CODE is not able to show a state of the application through the visual model. The same holds also for HeNCE that had similar features like CODE but expressiveness of its language was restricted. In GRADE, the application could be visually debugged but the visual language is based on different concept in comparison to our tool.

Despite that we are not aware of any such tool that is still actively developed or become widely accepted, we think that the visual approach to developing parallel applications is interesting and it deserve another chance. Parallel computers are more common and more accessible today; therefore, more scientific and engineering applications can profit from such hardware and not all of them require optimized handmade solutions. Additionally, we want to create a unified environment, where the same visual model is used not only during the development, but also to simplify various supportive activities.

A more successful approach to more abstract parallel programming is stream processing (StreamIt [12], FumeJava, Fastflow, etc.). FlumeJava [6] can be described as follows. It is a Java library for developing data-parallel pipelines. It offers lazy colle ction types and operations (map, filter, count, etc.), automatically generating and running a sequence of MapReduces only when the actual results are requested. MapReduce offers a good abstraction where many real parallel problems can be expressed. But data-flow is inherently limited by the target MapReduce framework – all data are processed uniformly in alternating and isolated map and reduce steps. From the perspective of these tools our tool offers a more low-level approach, less abstract programs. and more control over final applications. In our approach, we want to offer a more flexible environment where the user has more control to experiment with parallel algorithms.

Another approach is to introduce special constructions or annotations into widely used languages. As an example, we can name OpenMP[2], Unified Parallel C[3] [5] or Cilk++ [10]. The combination with standard languages makes these frameworks good prototyping tools, because a sequential program can be gradually parallelized. Many standard patterns, like parallelization of an independent for-cycle, can be also easily expressed in these tools. Our approach may

[2] http://openmp.org/wp/
[3] http://upc.lbl.gov/

need more work in the initial phase, because drawing a visual model is more demanding than setting up some annotations. But such model is useful for clear representation of the inner state of a running application, so it can speed up understanding of the application's behavior during testing, profiling, and other supportive activities.

If standard algorithms from a specific area are needed it is often the best solution to use some specialized libraries with tuned implementations. Considering numerical computations, there are specialized libraries for parallel numerical computations - for example libraries PETSc[4] and Trilinos[5]. However, when some special needs are required, it can be hard to adjust these libraries. They can be good prototyping tools and they can solve different problems, but if we want to experiment with different parallelization approaches, then they are usually not sufficient.

Use of such libraries is compatible with our approach. It is possible to combine their sequential parts with Kaira. For example, considering numerical computations, Kaira controls the overall data flow and Trilinos matrices are used for computations themselves.

3 Tool Kaira

This section serves as an overview for our tool Kaira; for more details see [1,2]. Our goal is to simplify the development of Message Passing Interface (MPI)[6] parallel applications and create an environment where all activities (prototyping, debugging, performance prediction, profiling, etc.) are unified under one concept.

The key aspect of our tool is the usage of visual models. In the first place, we have chosen visual models to obtain an easy and clear way to describe and expose parallel behavior of applications. The other reason is that a distributed state of an application can be shown through such visual model. The representation of an inner-state of distributed applications by a proper visual model can be more convenient than traditional ways like back-traces of processes and memory watches. Using this approach, we provide visual simulations where the user observes the behavior of the developed application. It can be used on incomplete applications from an early stage of development; therefore, it is a very useful feature for a prototyping tool. In a common way of development of MPI programs, it often takes a long time to get the developed application to a state where its behavior can be observed. We also use the same visual model for debugging and profiling. The user specifies through the visual model what to measure and Kaira generates a tracing version of the application recording its own runs. The record is presented back to the user through the original visual model of the application.

On the other hand, we do not want to create applications completely through visual programming. Sequential parts are written in the standard programming language (C++) and combined with the visual model that catches *parallel*

[4] http://www.mcs.anl.gov/petsc/
[5] http://trilinos.sandia.gov/
[6] http://www.mpi-forum.org/docs/docs.html

aspects and *communication*. We want to avoid huge unclear diagrams; therefore, we visually represent only what is considered as "hard" in parallel programming. Ordinary sequential codes are written in a textual language. Moreover, this design allows for easy integration of existing C++ codes and libraries.

It is important to mention that our tool is *not* an automatic parallelization tool. Kaira does not automatically detect parallelizable sections. The user has to explicitly define them, but they are defined in a high-level way and the tool derives implementation details.

Semantics of the Kaira visual programming language is based on Coloured Petri nets (CPNs) [8]. Petri nets are a formalism for description of distributed systems. They also provide well-established terminology, natural visual representation of models for their editing, and their simulations. The modeling tool *CPN Tools*[7] inspired us how to show visual models.

To demonstrate how our models work, let us consider a model in Fig. 1. It presents a problem where some jobs are distributed across computing nodes and results are sent back to process 0. When all these results arrive, they are written into a file. Circles (*places* in terminology of Petri nets) represent memory spaces. Boxes (*transitions*) represent actions. Arcs run from places to transitions (*input arcs*) or from transitions to places (*output arcs*). When a transition is executed it takes values (*tokens*) from places according to input arcs. When a computation in a transition is finished, then it produces new tokens to places according to output arcs. A computation described by this diagram runs on every process. Transferring tokens between processes are defined by the expression followed after character "@" in expressions on output arcs. A double border around a transition means that there is a C++ function inside. It is executed whenever the transition is fired. A double border around a place indicates an associated C++ function that defines the initial content of the place.

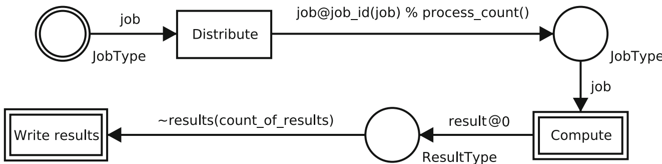

Fig. 1. The example of a model

4 Libraries

The infrastructure of libraries in Kaira is based on modules. A *module* is a model in Kaira enriched by an interface (depicted as a gray rectangle around the model). From a set of modules, Kaira generates a C++ library. As an example, consider the module in Fig. 2. It takes two input integers (x, y) and outputs a

[7] http://cpntools.org/

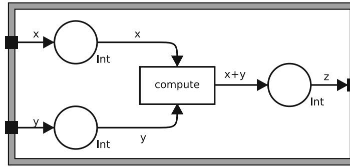

Fig. 2. The module *sum*

single integer (z). The example of a bigger module is presented in Sect. 5. When a library is generated from this module, we obtain the C++ library with the following interface:

```
void calib_init(int argc, char **argv);
void sum(int &x, int &y, int &z);
```

It is possible to use such library in any sequential C++ application. This application can be compiled and run through MPI infrastructure. The application will be executed in process 0 sequentially. When it calls a generated function, then the computation will be run across all MPI processes according to the structure of the module. When the function is finished (i.e. the module is finished) then the program continues again sequentially.

The library can be also generated in the *Remote Procedure Call* (RPC) mode. Kaira generates both server and client parts. The client side is a library that has the same interface as was described in the example above, but when a function is called it sends a request through a network to the server where a requested computation is executed.

4.1 Octave Libraries

Octave offers a possibility to create C++ modules (so called "oct-files"). It makes calling C++ functions accessible in the Octave environment. We use this infrastructure; Kaira is able to generate an oct-file that wraps our parallel libraries. Hence, the user is able to use modules smoothly in Octave in a similar way as in C++ applications. The important aspect of such integration is interoperability between data types. Kaira contains conversion functions for basic data types likes numbers or vectors. The user must provide conversion functions for own data types.

5 Case Study

As an example we have chosen a variant of the Finite Element Tearing and Interconnecting (FETI) domain decomposition method – Total-FETI [4][8]. Omitting other aspects like numerical scalability; parallelization of Total-FETI can be very

[8] The model and source codes used in this example are available on the website of our project http://verif.cs.vsb.cz/kaira.

straightforward. In [4], the basic idea is to decompose the domain into N subdomains. After the discretization, we get a block diagonal stiffness matrix (Eq. 2 in [4]), where matrices $\mathbf{K}_1 \dots \mathbf{K}_N$ are stiffness matrices for corresponding subdomains. Using such block diagonal stiffness matrix $\mathbf{K}$, we are usually able to divide computations and perform them in parallel (see the following equation).

$$\mathbf{K} = \begin{bmatrix} \mathbf{K}_1 & & \\ & \ddots & \\ & & \mathbf{K}_N \end{bmatrix}, \mathbf{x} = \begin{bmatrix} \mathbf{x}_1 \\ \vdots \\ \mathbf{x}_N \end{bmatrix}, \mathbf{Kx} = \begin{bmatrix} \mathbf{K}_1\mathbf{x}_1 \\ \vdots \\ \mathbf{K}_N\mathbf{x}_N \end{bmatrix}. \quad (1)$$

Such straightforward approach is far from being optimal. More advanced approaches were published in [7], where the authors focus on performance and usage of thousands of processors. In their solution, they use the library PETSc. Of course, such solution is much more time and resource demanding and relatively very complicated. On the other hand, Octave implementation that we start with roughly follows steps from paper [4] and it is relatively simple and readable. With such implementation, it is easy to explore different mathematical aspects or perform different experiments, but it is hard to address issues related to parallel programming.

More precisely, in the Octave API, there are external packages[9] for parallel/distributed computing. Package *general* contains two functions (`parcellfun` and `pararrayfun`) that evaluate functions using multiple processes. But it is restricted only to shared memory architectures. For distributed memory there are packages *openmpi_ext* and *parallel*. These packages are basic wrappers to MPI functions and simple sockets API. In both cases, they are quite low-level interfaces from a programmer's point of view. Tasks like debugging and profiling can be complicated considering development environments for Octave.

At the beginning, we had a working sequential implementation of Total-FETI for Octave. The most time consuming operation is a solution of linear system $\mathbf{Ky} = \mathbf{x}$. As was suggested in [4], Cholesky factorization of the stiffness matrix is used (`[L,ans,P]=chol(K,'lower')`). The following Octave code performs this time consuming computation:

```
function y=Kplus_aux(L, P, x)
    Lt=L';
    Pt=P';
    y=P*(Lt\(L\(Pt*x)));
end
```

Total-FETI iteratively computes the result and thus uses `Kplus_aux` several times. We want to parallelize this operation using Kaira and explore its properties. Module `Kplus_par` from the Fig. 3 defines the parallel computations. In this model we use types *Matrix* and *SparseMatrix* which are native types offered by the Octave C++ interface.

Matrices $\mathbf{L}$ and $\mathbf{P}$ are a block diagonal. First, they are (along with vector $\mathbf{x}$) divided according to their block diagonal structure (transition *Divide*). By

[9] All mentioned packages are available at http://octave.sourceforge.net

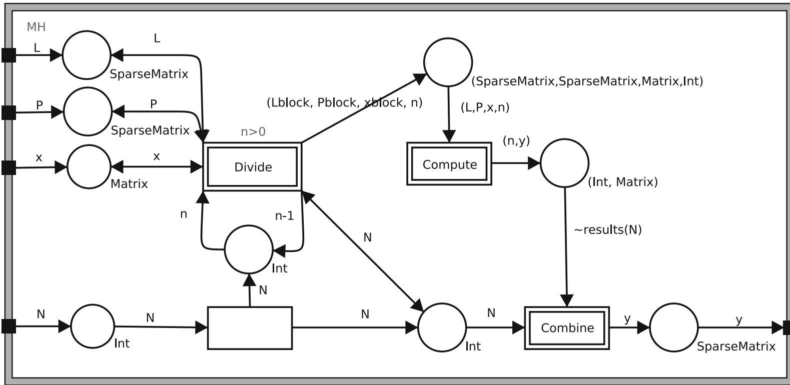

Fig. 3. Module *Kplus_par*

this action we obtain N smaller tasks. These tasks are processed by transition *Compute*. A source code in the transition *Compute* performs the same computation like the original Octave function `Kplus_aux`, but using a single block that represents one sub-domain. The following source code is stored in the transition *Compute*. The user needs to write only the three lines in the body of the function. The rest is a template generated by the tool.

```
struct Vars {
    Matrix x; Matrix y;
    SparseMatrix L; SparseMatrix P;
    int n;
};
void transition_fn(CaContext &ctx, Vars &var) {
    SparseMatrix Lt=var.L.transpose();
    SparseMatrix Pt=var.P.transpose();
    var.y = var.P*(Lt.solve(var.L.solve(Pt*var.x)));}
}
```

When all partial results are produced, transition *Combine* is fired and the resulting vector is composed. After that, the module is terminated and the data are transmitted back to the Octave. In the original source code for Octave, the only change is the call of the generated function (instead of the original sequential one). It has one additional parameter N indicating the number of sub-domains.

The model shown in Fig. 3 represents a solution for a shared memory system. If the resulting application is started in a configuration with multiple threads, then it is performed in parallel. An extension of this net for usage with distributed memory is easy. We just need to modify the existing arc inscription: *(Lblock, Pblock, xblock, n)* to: *(Lblock, Pblock, xblock, n)@n* and *(n, y)* to *(n, y)@0*. It causes that blocks will be assigned to MPI processes according to their positions and results are sent back to process 0.

5.1 Experiments and Results

Now with the existing model, a user is able to use various features of Kaira. For example it is possible to perform simulations of module executions independently on Octave code (Fig. 4), where the user manually controls the simulation. Additionally, it is possible to run the application in the tracing mode where the execution of a module is recorded. Such recorder execution can be replayed using the original model or some performance statistics can be obtained (like execution times for each transition, etc).

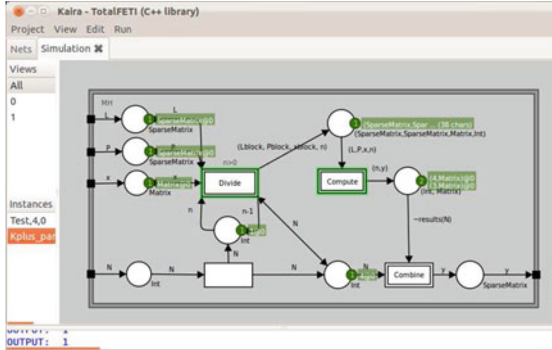

Fig. 4. A screenshot of a simulation

To test the module's performance, we solve a displacement of a 1D string that is fixed on both ends. We prepare a stiffness matrix where each sub-domain has 500000 discretization steps and we use 30 sub-domains. The measurement was performed on a computer with 8 processors AMD Opteron/2500 (having a total of 32 cores). A computation of original `Kplus_aux` takes in average 21.79 s in the pure Octave solution. We measured runs of the library generated from our module `Kplus_par`. The test was performed in RPC mode, where both client and server run on the same computer. The measured times for multithreading and MPI backends are listed in Table 1.

These results are consistent with reasonable expectations. They show that there is a communication cost related to the RPC mode (difference between running times for multithreading with and without RPC). This cost is fixed due

Table 1. Running times (in seconds) of *Kplus_par* using threads and MPI

Nodes	1	2	4	8	16	32
Threads + RPC	25.30	19.87	10.32	8.02	7.70	7.76
MPI + RPC	25.19	20.88	16.89	15.36	16.55	16.39
Threads (no RPC)	21.71	11.05	6.51	4.35	4.06	3.93

to the fixed problem size. It also presents a bottleneck for further performance improvements. For multithreading we reached this bottleneck around 16 cores. At this point, a time to distribute matrices is much bigger than a time to perform the computation itself. Usage of MPI introduces additional communication overhead, because data are distributed between nodes using MPI functions.

The execution of the sequential version (using only one core) of the whole computation takes approximately 320 s while the function `Kplus_aux` was used 5 times. For real problems, the number of iterations can be different (usually bigger) and when the stiffness matrices contain more non-zero elements, the function `Kplus_aux` will be more time consuming. We present these results mainly to prove, that we are able to get a working parallel solution with reasonable performance and even if the obtained solution may not be the most optimal, we were able to develop it fast.

To keep the presented solution simple, the stiffness matrix $\mathbf{K}$ is divided and distributed for every computation. To improve the performance further, we can store these blocks in computing nodes and use them several times, while they do not change during the computation. Mentioned matrices are sparse, but they are usually huge. Their size is based on the number of primary variables and there can be millions of primary variables in real problems. So for real experiments the memory consumption often becomes an issue (usually even before the performance). In fact, we do not need to compose the whole stiffness matrix. When we divide its blocks between computing nodes, we can handle a problem originally too large for a single computer. This can be an even bigger advantage than just the performance improvement.

6 Conclusion

The paper presents our tool Kaira and its ability to generate parallel libraries. Also it demonstrates their usage in a combination with Octave. Such combination allows rapid prototyping of parallel numerical computations. On a real example (TotalFETI method), we demonstrated that it takes only a few lines of C++ code and a relatively simple model based on CPNs to get a working parallel application with reasonable performance. Further, this model can be easily extended and we are use the same visual model for debugging and profiling. Thus, an inexperienced user does not have to learn additional tools.

Kaira is still being actively developed. We are trying to improve the process of gathering information from the model to provide functions like performance prediction or verification. From the perspective of this paper, we are also focused on additional simplifications for binding Octave types and allow for preserving data on computing nodes during consecutive computations. We also want to extend our approach to Matlab.

Acknowledgments. The work is partially supported by: GAČR P202/11/0340, the European Regional Development Fund in the IT4Innovations Center of Excellence project (CZ.1.05/1.1.00/02.0070) and Grant of SGS No. SP2013/145, VŠB - Technical University of Ostrava, Czech Republic.

References

1. Böhm, S., Běhálek, M.: Generating parallel applications from models based on petri nets. Adv. Electr. Electron. Eng. **10**(1), 28–34 (2012)
2. Böhm, S., Běhálek, M.: Usage of Petri nets for high performance computing. In: Proceedings of the 1st ACM SIGPLAN Workshop on Functional High-Performance Computing, FHPC '12, pp. 37–48. ACM, New York (2012) http://doi.acm.org/10.1145/2364474.2364481
3. Browne, J.C., Dongarra, J., Hyder, S.I., Moore, K., Newton, P.: Visual programming and parallel computing. Technical report, Knoxville, TN, USA (1994)
4. Dostál, Z., Horák, D., Kučera, R.: Total FETI-an easier implementable variant of the FETI method for numerical solution of elliptic PDE. Commun. Numer. Meth. Eng. **22**(12), 1155–1162 (2006). http://dx.doi.org/10.1002/cnm.881
5. El-Ghazawi, T., Smith, L.: UPC: unified parallel C. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, SC '06. ACM, New York. http://doi.acm.org/10.1145/1188455.1188483 (2006)
6. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. SIGPLAN Not. **41**(11), 151–162 (2006)
7. Horák, D., Dostál, Z.: Parallelization of the total-FETI-1 algorithm for contact problems using PETSc. In: Proceedings of the Second International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering. Civil-Comp Press, Stirlingshire (2011)
8. Jensen, K., Kristensen, L.M.: Coloured Petri Nets - Modelling and Validation of Concurrent Systems. Springer, Heidelberg (2009)
9. Kacsuk, P., Cunha, J., Dózsa, G., Lourenco, J., Fadgyas, T., Antao, T.: A graphical development and debugging environment for parallel programs. Parallel Comput. **22**(13), 1699–1701 (1997). http://www.sciencedirect.com/science/article/pii/S0167819196000750 (distributed and parellel systems: Environments and tools)
10. Leiserson, C.: The Cilk++ concurrency platform. J. Supercomput. **51**(3), 244–257 (2010). http://dx.doi.org/10.1007/s11227-010-0405-3
11. Newton, P., Browne, J.C.: The code 2.0 graphical parallel programming language. In: Proceedings of the 6th International Conference on Supercomputing, ICS '92, pp. 167–177. ACM, New York. http://doi.acm.org/10.1145/143369.143405 (1992)
12. Thies, W., Amarasinghe, S.: An empirical characterization of stream programs and its implications for language and compiler design. In: Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques, PACT '10, pp. 365–376. ACM, New York (2010)