parXXL: A Fine Grained Development Environment on Coarse Grained Architectures

Jens Gustedt, Stéphane Vialle, Amelia De Vivo

parXXL: A Fine Grained Development Environment on Coarse Grained Architectures

Jens Gustedt[1], Stéphane Vialle[2], and Amelia De Vivo[3]

[1]INRIA Lorraine & LORIA, France
[2]SUPELEC, France
[3]Università degli Studi della Basilicata Italy

Abstract. We present a new integrated environment for cellular computing and other fine grained applications. It is based upon previous developments concerning cellular computing environments (the ParCeL family) and coarse grained algorithms (the SSCRAP toolbox). It is aimed to be portable and efficient, and at the same time to offer a comfortable abstraction for the developer of fine grained programs. A first campaign of benchmarks shows promising results on clusters and mainframes.

1 Project overview

Nowadays, designers and developers of algorithms and code for large scale applications are often confronted with a paradoxal situation: their modelling and thinking is *fine-grained*, speaking *e.g.* of atoms, cells, items, protein bases and alike, whereas modern computing architectures are *coarse-grained* providing few processors (up to several thousands $\approx 10^3$) to potentially huge amount of data (thousands of billions of bytes $\approx 10^{12}$) and linking a substantial amount of resources (memory in particular) to each processor. Only few tools (for both, modelling and implementation) are provided to close this gap in expectation, competence and education.

On the modelling side, Valiant's seminal paper on the BSP, see [1], has triggered a lot of work on different sides (modelling, algorithms, implementations and experiments) that showed very interesting results on narrowing the gap between, on one hand, fine grained data structures and algorithms and, on the other, coarse grained architectures. But when coming to real life code developers are usually left alone with the "classical" interfaces, even when they implement with a BSP-like model in mind. In particular, implementing dynamic data structures such as *cellular networks* efficiently on a large scale often constitutes an insurmountable hurdle for real life applications.

The parXXL development environment is split into several, well-identified layers which historicaly come from two different project sources, SSCRAP and ParCel-6. Its software architecture is introduced on Fig. 1, and demonstrates the split of these two main parts into the different layers. The (former) SSCRAP part introduces all the necessary parts to allow for an efficient programming in coarse grained environments; interfaces for the C++ programming language, the POSIX system calls, tools for benchmarking, a memory abstraction layer and the runtime communication and control. The (former) ParCeL-6 part introduces a *cellular* development environment and a set of predefined and optimized cell networks. These programming models of SSCRAP and ParCeL-6 are detailed in the next sections.

2 SSCRAP programming model

SSCRAP is a programming environment that is based on an extension of the BSP programming model [1], called PRO [2]. It has proven to be quite efficient for a variety of algorithms and platforms, see [3]. Its main features what concern this paper are:

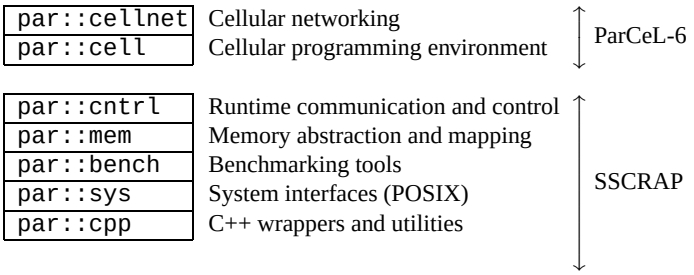

Fig. 1. parXXL software architecture

Supersteps with relaxed synchronization: Originally, BSP was designed with strong synchronization between the supersteps. PRO (and thus SSCRAP) allows a process to resume computation as soon as it receives all necessary data for the next superstep. The `par::cntrl` layer (see Fig.1) implements these features in parXXL.

A well identified range of applicability: SSCRAP is clearly designed and optimized for *coarse grained* architectures, *i.e.* where there is substantially more memory than there are processors.

Comfortable encapsulation of data: The work horse of SSCRAP is a data type (`chunk` in the `par::mem` layer) that encapsulates data situated on different supports such as memory and files which then can be mapped efficiently into the address space of the processes. Thereby SSCRAP can efficiently handle huge data (*e.g* larger than the address space) without imposing complex maintenance operations to the programmer.

Portability: SSCRAP is uniquely based on normalized system interfaces (`par::sys`), most important are POSIX file systems, POSIX threads and MPI for communication in distributed environments. Therefore it should run without modification on all systems that implement the corresponding POSIX system calls and/or provide a decent MPI implementation.

Performance: This portability is *not* obtained by trading for efficiency. In the contrary, we provide two run-times, one for shared memory architectures (threads) and one for distributed computing (MPI). These are designed to get the best out of their respective context: avoiding unnecessary copies on shared memory and latency problems when distributed. All this is achieved by only linking against the respective library, no recompilation is necessary.

3 ParCeL-6 programming model

The `par::cell` level of parXXL architecture (see Fig. 1) implements the ParCeL-6 programming model [4]. It is based on *cells* distributed on different processors, and is an *extended cellular model*:

A sequential program with cellular operations: ParCeL-6 developers design and implement some cell behavior functions, and a sequential program to install and to control a parallel cellular net. This mixed programming model is easy to use and facilitates *cellular servers* designs: a classical client can connect to the sequential program, that runs cellular computations.

A dynamic cellular network: Starting from an empty network of cells, the sequential program creates cells on all available processors. Each cell has an individual set of parameters, and the first action of these cells is usually to connect each other to create a cellular

network. This network may evolve at any point of the execution: cells and connexions can be created or removed.

Six cell components: A cell is composed of (1) a unique *cell registration*, (2) *parameters*, (3) *private variables*, (4) some *cell behavior functions*, (5) a unique multi valued *output channel*, and (6) several multi valued *input channels*. The first is imposed by ParCeL-6 mechanisms, the others are defined by the developer.

A cyclic/BSP execution of the cell net: A ParCeL-6 cycle consists of three steps: computation, net evolution and communication. Each cell is activated once during the computation step, where it sequentially reads its inputs, updates its output, and issues some cell net *evolution requests*. These requests define, kill, connect or disconnect some cells, and are executed during the net evolution step.

Three modes of cellular communications: Cells need to be explicitly connected to communicate, and a cell output can be connected to an unlimited number of cell inputs. During the communication step, *buffered outputs* are copied to their connected cell inputs. Their propagation is fast and is adapted to synchronous fine grained computation (cell inputs do not change during the computation steps). The propagation of a *direct output* to a connected cell input is triggered each time a *refresh* command is executed for it. This mechanism has a large overhead but is required by some asynchronous fine grained computations [4]. *Hybrid outputs* are an attempt to get both fast and asynchronous cellular computations: they propagate their value one time per computation step and per processor (cells on different processors can read different values during one computation step).

Global communications with the cell net: Some *global* cell net communications mechanisms allow the sequential program to send input data to the cells (like camera images) and to obtain the cell outputs (computation results).

4 Optimized cell network library

The `par::cellnet` library (see Fig. 1) is a collection of *cell network installers*: application code can easily deploy a cell network just using an *installer* object. Each *installer* has to be set with the application cell behavior functions, the cell parameter and cell variable types, and the cell network size. Then, it installs the cells and their parameters, and connects the cells according to a predefined communication scheme. Deployments are optimized: cells are load balanced among the processors but neighbor cells are installed on a same processor, and cell net installation is split in small steps to limit the memory required by the deployment operations. The `par::cellnet` library currently includes 3D cubic and 2D matrix networks *installers*.

5 Application and performance examples

To validate the scalability of parXXL we have designed and experimented a 3D Jacobi relaxation on a *cube* of cells. The cells are created with one output value, and are connected to their neighbor cells: up to six neighbors for a cell inside the cube. Then, the parXXL program enters a long loop of cellular execution cycles. Cells inside the cube update their output value with the average of their neighbor output values, while cells on the cube boarder maintain their output value unchanged. To deploy large cubes of cells, we have used the `par::cellnet` library, that installs optimized cell networks and require small datastructures for the cell creation management.

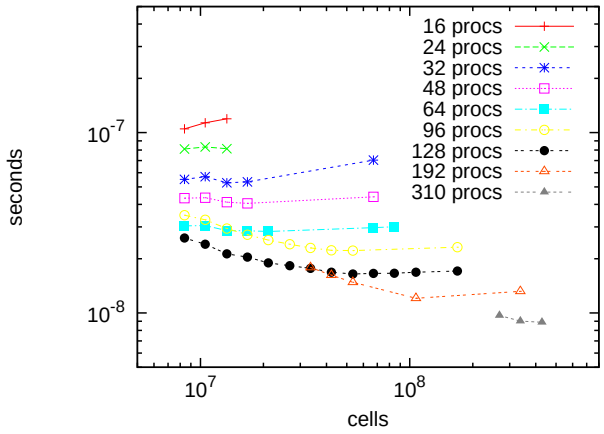

Fig. 2. Benchmarks on Grid-eXplorer cluster: up to 400 million cells using up to 310 processors

Fig. 2 illustrates the execution time measured per cell and per cycle of our 3D Jacobi relaxation algorithm. The platform is the *Grid-eXplorer* cluster composed of bi-processor machines. We have computed problems up to 400 million cells and used up to 310 processors in total. We can see this parXXL application *scales* well with the number of processors: (1) the *cell execution time* decreases substantially when running a fixed number of cells on an increasing number of processors, and (2) the *cell execution time* remains constant or continues to decrease when running more cells on more processors (see the curves for 128, 192 and 310 processors).

Moreover, first experiments on a SGI-Origin3000 have shown parXXL environment also runs and achieves speedup on a mainframe with a proprietary ccNUMA architecture.

6 Conclusion and perspectives

Main parts of parXXL architecture are implemented and operational, and first experiments show parXXL architecture scales up to 310 processors on a large fine grained application. Next development steps will consist in implementing the *global communication mechanisms* between the cell net and the sequential program, and to design and implement an efficient *hybrid cell communication mode*. Next experiment will be run on a larger number of processors of the Grid-eXplorer machine, and on a Grid of clusters using Grid5000 (the French experimental Grid).

From an application point of view, we aim to run 10^9 cell simulations of optical phenomena in 2006, in collaboration with researchers from LMOPS laboratory.

References

1. Valiant, L.: A bridging model for parallel computation. Communications of the ACM **33**(8) (1990)
2. Gebremedhin, A., Guérin Lassous, I., Gustedt, J., Telle, J.: PRO: a model for parallel resource-optimal computation. In: 16th Annual International Symposium on High Performance Computing Systems and Applications. (2002)
3. Essaïdi, M., Gustedt, J.: An experimental validation of the PRO model for parallel and distributed computation. In: 14th Euromicro Conference on Parallel, Distributed and Network based Processing. (2006)
4. Ménard, O., Vialle, S., Frezza-Buet, H.: Making cortically-inspired sensorimotor control realistic for robotics: Design of an extended parallel cellular programming models. In: International Conference on Advances in Intelligent Systems - Theory and Applications. (2004)