ParCeL-1: a parallel programming language based on autonomous and synchronous actors

Stéphane Vialle
Supélec
2 rue Edouard Belin
F-57078 Metz cedex 3
vialle@ese-metz.fr

Thierry Cornu
Swiss Federal Institute of Technology
EPFL DI, LITH
CH-1015 Lausanne
cornu@di.epfl.ch

Yannick Lallement*
Crin-CNRS-Inria Lorraine
BP 239
F-54506 Vandoeuvre-les-Nancy
yannick@cmu.edu

Abstract

In this paper we present a new language, called ParCeL-1, intended to make easier the implementation of computation-intensive applications on highly parallel MIMD architectures. The computational model of ParCeL-1 is object-oriented and synchronous. The current prototype is implemented on several multi-processor systems (Cray T3D, Intel Paragon, Telmat T-Node and workstation networks using PVM). Benchmarks are presented for several parallel programs.

1 Introduction

For the last decade, the main stream of research in parallel programming languages has focused around two main directions: message passing and data parallel programming models. In message passing models, concurrent sequential processes communicate via explicit send and receive instructions. In data parallel models, compilers provide automatic or semi-automatic distribution of data, as well as parallelization of loop constructs.

Message passing is the most general way of programming MIMD machines. It usually results in more efficient implementations, especially on massively parallel machines that are too large to use physically shared memory. However, it requires from users a low-level description of the implementation. Especially, parallelization, data partitioning and communications must be specified explicitly.

Conversely, data parallel models are comparatively easier to program: tedious tasks like the distribution of code and data structures is, to some extent, automated. Also, already existing sequential code is easy to port to data parallel languages. Unfortunately, parallelizing compilers for data parallel languages currently handle only loop nests. Moreover, they work efficiently mainly with regular data structures without indirect addressing; reordering data between two successive loops is an advanced research topic at the time of writing. Data parallel programming models are therefore suitable mainly for scientific computation and only to solve regular problems.

Between low-level but general message passing models on one hand and data parallel models on the other hand, easier to use but restricted to one category of problems, there is clearly a need for intermediate approaches. ParCeL-1 is aimed to provide an alternate programming model that would:

- make it possible to program a wider class of applications than data parallel languages;
- still partly hide to the programmer the architecture of the parallel machine;
- yield acceptable performance.

ParCeL-1 [16] hides to the user the actual parallel architecture and is efficient on a wide variety of

*Currently at Carnegie Mellon University, Human-Computer Interaction Institute, 5000 Forbes Avenue, Pittsburgh PA 15213, USA applications (e.g. scientific computing, artificial intelligence, connectionism...). ParCeL-1 is based on an intrinsically parallel computing model which attempts to be a compromise between high-levelness and efficiency. In this paper, we first present this computing model ; then we present several sample applications and performances obtained on the Cray T3D multicomputer.

2 ParCeL-1 computational model

2.1 Other related programming models

Concurrent object-based programming

Many parallel programming models have been proposed in the past years as alternatives to message passing or data parallel programming. Among the most prolific class of alternate models are those based on concurrent objects or actors. Concurrent object computation traces back to the actor model proposed by Hewitt [9] and later improved by Clinger and Agha [1]. The most recent system in this lineage is the HAL [10] language. The concept of actors has also been developed independently by the team of Yonezawa since the mid-eighties, yielding the successive variants of the ABCL language [17, 13]. In parallel to actor languages, extensions of compiled object languages like C++ or Eiffel also introduced concurrent objects [4, 8].

In actor- or object-based models, parallelism of execution is expressed explicitly. The main specificity of these models is that they make programmers write already distributed code. This opens interesting opportunities for automated data distribution and process mapping. Nearly all models based on concurrent objects or actors use asynchronous execution models, i.e., models where computation and message sendings are not synchronized. Consequently, communication has to be synchronized explicitly and each transaction between two objects must be implemented as a separate communication. Mail boxes or message queues have to be managed by the underlying system. As a result, communication is not easy to implement efficiently and concurrent object systems are often restricted to coarse grain parallelism for performance reasons. As a result, concurrent object programming, while finding a wider and wider acceptance for implementing distributed systems over wide area networks [5], is still seldom used in massively parallel high performance computers.

Bulk-synchronous computation

Bulk-synchronous computation was first introduced by Valiant [14, 15] with the Bulk Synchronous Programming (BSP) model. Computation in a BSP program consists of a sequence of super-steps; each super-step is a sequence of steps followed by a synchronization barrier at which non-local communications take effect. BSP provides a framework for automatic data allocation, for latency hiding and for performance prediction. Theoretical results [15] strongly suggest that it may be implemented efficiently on many parallel computer architectures. In the recent years, actual programming tools using BSP were developed [7, 3]. At the time of writing, a standardization effort is going on to set up a worldwide standard for a BSP library [6].

Meanwhile, new MIMD architectures appeared such as the Cray T3D and T3E, in which virtual shared memory as well as synchronization barriers are hardware-implemented. The most efficient way to program these machines is to use asynchronous remote memory access, also called one-sided message passing, and synchronization barriers. This programming style closely matches the BSP model and implementing a BSP library on top of these machines is extremely simple [12]. Introduction of one-sided message passing in standard parallel programming models like High Performance Fortran is also under discussion.

The simultaneous development of BSP programming tools and of new generations of hardware with virtual shared memory contributes to making bulk synchronism an emerging universal programming model for MIMD architectures.

2.2 Cell computational model

The computational model of the ParCeL-1 language is inspired from actor- and object oriented programming models; however it borrows from the BSP model the idea of bulk synchronism. Combining an actor-

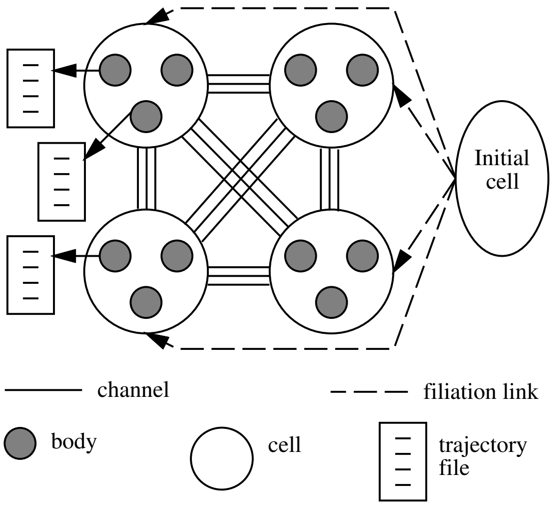

Figure 1: The N bodies implementation

based approach with synchronism is the original feature of the computational model.

The concurrent objects in ParCeL-1 are called *cells*. They are dynamically created instances of cell types declared before run-time.

The communication graph between cells is explicit: two cells can communicate only if a *communication channel* has previously been established between them. The communication graph made of the cells and their connecting channels is reconfigurable at run-time.

Computation is a sequence of cycles somewhat similar to the super-steps of a BSP machine. A cycle is made of three main phases: computation, management and communication. In the computation phase, each cell updates the values of its output channels. It may also emit requests to modify the cell network topology, i.e., to create or destroy cells, or to create new connections. These requests are executed in the management phase. In the communication phase, the new values of communication channels are broadcasted so as to be used in the next computation cycle.

3 Example of application

As an example of ParCeL-1 program, we present here a possible implementation of a computation-intensive application: the N bodies problem. This problem consist in determining the trajectories of N bodies in mutual interaction (e.g. through gravitation), each body being influenced by all the other ones. The computing method is iterative. At each time step, the acceleration of all bodies is computed based on the position of all other bodies and on the law of interaction. An approximation of the speed and then of the displacement of the bodies can be computed, yielding their positions at the next iteration.

In this ParCeL-1 implementation, we chose to associate one cell to each body. The cells are fully interconnected so that each body can be aware of the position of all the other ones (figure 1). The cellular graph is established by an initial cell (called `main`), and does not change afterwards. Figure 2 shows the source code of the `main` cell type. This cell type consists of a set of local variables and a set of `condition ==> action` rules that govern the behavior of the cell. The creations and connections of the `Body` cells are requested by the first rule; they are executed during the first cycle of the program.

Figure 3 shows the source code of a `Body` cell that computes the new position and speed of its associated body in a two-dimensional space, from the data sent by all the other cells. This cell is composed of the same elements as the `main` cell, plus a set of input and output channels (variables labeled `in` and `out`), and a local function (`InitSys`). The core work in done in the second rule of the cell, that is executed until the desired number of iterations is reached. The trajectories of the bodies are stored in a set of files, one per `Body` cell. These files can be accessed in parallel when their host cells are executed on different processors. This example illustrates the parallel I/O capabilities of ParCeL-1, provided that the underlying system (hardware and operating system) allows real parallel access to several files or disks; otherwise the parallel I/O of ParCeL-1 will be serialized by the underlying system.

When this program is run, the cell distribution on the processors, the load-balancing and the synchronization are taken care of by ParCeL-1. The parallel performance of this application on Cray T3D is given in the next section.

```
typecell main() {
  immat Body TabBody[N];             /* References of the Body cells         */
  int i, cconn, ccell, citer;        /* Connections, cells, iterations counters*/

  initially ==> {                    /* Rule executed on creation of the cell */
    for (i = 0; i < N; i += 1)       /* Creation of the Body cells           */
      TabBody[i] = create Body(NB_ITER,i);
    citer = 0;
    for (cconn = 0; cconn < N; cconn += 1)  /* Connection between these cells  */
      for (ccell = 0; ccell < N; ccell += 1) {
          connect PosIn[cconn][0] of TabBody[ccell]
               to PosOut[0] of TabBody[cconn];
          connect PosIn[cconn][1] of TabBody[ccell]
               to PosOut[1] of TabBody[cconn];
      }
  }
  TRUE ==> {                         /* Wait until all the iterations are    */
    citer += 1;                      /* completed                            */
    if (citer > NB_ITER) halt;
  }
}
```

Figure 2: Source code of the `main` cell of the N-bodies program

```
typecell Body(int NbIter, int me) {
 in double PosIn[N][2];              /* Current positions                    */
 out double PosOut[2];               /* Future positions                     */
 double SPE[2],WEI[2];               /* Speed, weight                        */
 double dp[2];                       /* Position differences buffer          */
 double a[2];                        /* Acceleration buffer                  */
 double d2;                          /* Square of the distance               */
 int i,iter;
 symbol Fs;                          /* Trajectory file                      */

 void InitSys(int i)                 /* Initialization function              */
 {
   /* ... */
 }
 initially ==> {                     /* Rule executed on creation of the cell */
   InitSys(me);
   Fs = mfopen('traject',me,'w');
   fprintd(Fs,PosOut[0]); fprints(Fs,' ');
   fprintd(Fs,PosOut[1]); fprintnl(Fs);
   citer = 0;
 }
 citer < NbIter ==> {                /* Compute new speed/position of the body */
   a[0] = 0.0; a[1] = 0.0;           /* Initialize temporary buffers         */

   for (i = 0; i < N; i += 1)        /* Compute total strength               */
     if (i != me) {
       dp[0] = PosIn[i][0] - PosIn[me][0];
       dp[1] = PosIn[i][1] - PosIn[me][1];

       d2 = dp[0]*dp[0] + dp[1]*dp[1];
       a[0] += G*WEI[i]*(dp[0])/d2/sqrt(d2);
       a[1] += G*WEI[i]*(dp[1])/d2/sqrt(d2);
     }
   SPE[0] += DELTA_T*a[0];         /* Compute new speed                      */
   SPE[1] += DELTA_T*a[1];

   PosOut[0] = PosIn[me][0] + DELTA_T*SPE[0]; /* Compute new position        */
   PosOut[1] = PosIn[me][1] + DELTA_T*SPE[1];

   fprintd(Fs,PosOut[0]); fprints(Fs,' ');    /* Save position in file       */
   fprintd(Fs,PosOut[1]); fprintnl(Fs);

   citer += 1;                       /* This was one more iteration          */
 }
 citer == NbIter ==> {               /* Final rule of the cell               */
   fclose(Fs);
   citer += 1;
 }
}
```

Figure 3: Source code of the `Body` cell of the N-bodies program

4 Benchmarks criteria

In order to evaluate ParCeL-1 performance, a sequence of criteria is defined. The first criterion evaluates mono-processor performance. Successive criteria give more and more refined information on performance comparison.

4.1 Mono-processor performance

As the ParCeL-1 compiler generates C code, and as C is an efficient and well known sequential language, it seems logical to compare mono-processor execution of ParCeL-1 programs to the execution of corresponding C programs. Thus, we first measure the overhead of the automatic C code generation of ParCeL-1, compared to a manually implemented C program. We define the Slow-Down of level 0 as follows:

$$SD0 = \frac{T_{\mathsf{ParCeL-1}}(1)}{T_C(1)} \tag{1}$$

where $T_{\mathsf{ParCeL-1}}$ is the time to run a ParCeL-1 program on one processor and T_C is the best available implementation of the same program directly in C.

4.2 ParCeL-1 implementation scalability

The second stage of our sequence of comparison criteria consists of computing the speed-up of a ParCeL-1 program execution on P identical processors versus the execution on only one of these processors. In fact, this criterion measures the efficiency of the ParCeL-1 parallelization. The level-one speed-up is defined as:

$$SU1(P) = \frac{T_{\mathsf{ParCeL-1}}(1)}{T_{\mathsf{ParCeL-1}}(P)} \tag{2}$$

where $T_{\mathsf{ParCeL-1}}(I)$ is the execution time of the benchmarked program on I processors; $SU1$ is a measure of the scalability of a ParCeL-1 program.

4.3 Speed-up of ParCeL-1 programs

At the next stage of our sequence of criteria, we compare the execution time of parallel ParCeL-1 programs on P processors, to the one of sequential C programs. We define the level-two speed-up as:

$$SU2(P) = \frac{T_C(1)}{T_{\mathsf{ParCeL-1}}(P)} \leq P \tag{3}$$

This speed-up can be expressed as a function of the previous measures, combining equations 1, 2 et 3:

$$SU2(P) = \frac{SU1(P)}{SD0} \tag{4}$$

The $SU2$ performance measure is the most relevant one for an application user. An efficiency corresponding to this speed-up can also be defined as follows:

$$E2(P) = \frac{SU2(P)}{P} \tag{5}$$

It can be easily derived from equations 3, 4 and 5 the following relation that illustrates the slow-down of ParCeL-1 programs versus C programs:

$$E2(P) \leq \frac{1}{SD0} \tag{6}$$

5 Benchmarks

We evaluate the ParCeL-1 implementation on two computation-intensive programs (Jacobi relaxation and N-body problem), and on a tree exploration (N-queens problem). In each case we have measured the SD0, SU1, SU2 and E2 coefficients. In this section, we summarize the results obtained on a Cray T3D.

5.1 Jacobi relaxation

The Jacobi relaxation is an iterative computing algorithm used to solve some differential equation systems. For example, it can be used to compute the voltage distribution in a metallic plate, on which a known potential distribution is applied on the sides (figure 4). The plate is approximated by a grid of points, and at each step of the relaxation the potential of each point is updated by computing the average of its four direct neighbors. In this example, the voltage distribution is computed after approximatively 2000 iterations.

The ParCeL-1 program consists of a main cell that creates a complete network of computing cells. Each of them is in charge of a sub-grid of points and saves its local results in a separate file, as shown on figure 4.

With a grid of 256×256 points spread on 256 cells (256 points per cell), and 1000 relaxation steps, the evolution of the execution time is shown on figure 5.

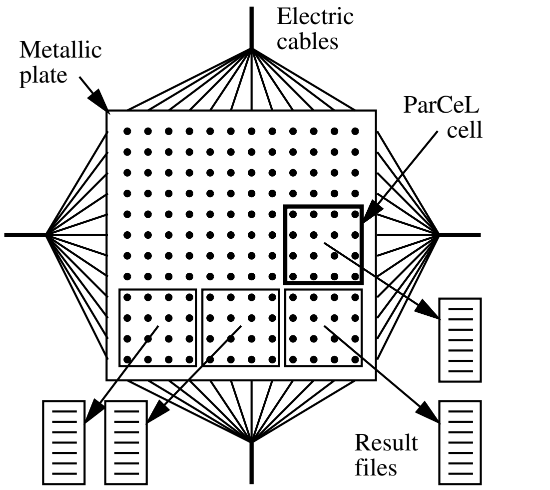

Figure 4: ParCeL-1 Jacobi relaxation implementation

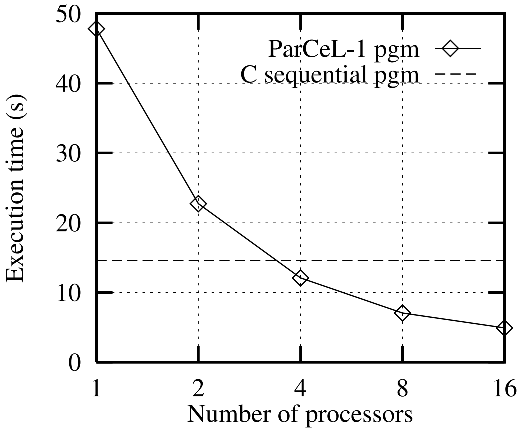

Figure 5: ParCeL-1 Jacobi relaxation execution time on a T3D multicomputer

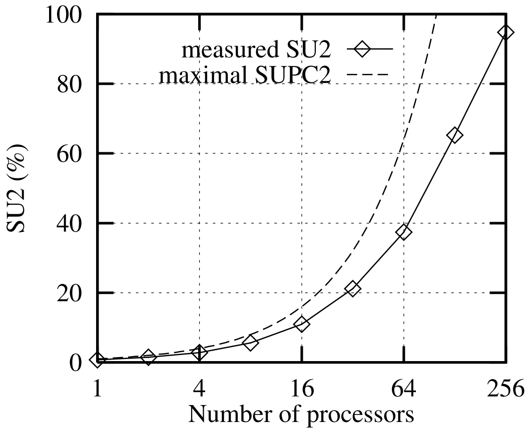

Figure 6: Speed-up of the 512-bodies problem on a T3D multicomputer, versus a sequential C program

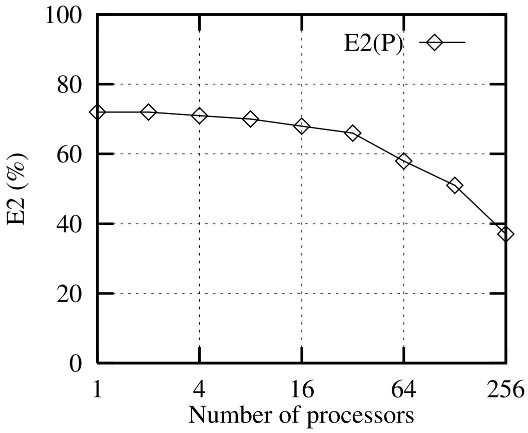

Figure 7: Efficiency of the 512-bodies problem on a T3D multicomputer, versus a sequential C problem

The parallel ParCeL-1 program is faster than the sequential C one, but the speed-up does not exceed 3, even on 16 processors. In fact, the Jacobi relaxation is unfavorable to a parallelization with ParCeL-1. First, grid point computations are very simple, and comparatively, the overhead for cell system management is not negligible. Therefore the sequential slow-down $SD0$ is extremely important in this example (nearly 5), limiting the effects of parallelization. Second, the cell connection pattern only requires local communications with neighbors, while the current prototype of ParCeL-1 implements systematic global communications for any kind of connection patterns. The communication overhead therefore limits the speed-up when the number of processors increases.

5.2 N-bodies problem

The N-bodies problem and its implementation in ParCeL-1 are the ones described in section 3.

N	Bodies per cell	SUPC2 (1000 iter)	EPC2 (1000 iter)
512	2	94.5	37%
1024	2	108.3	42%
1024	4	133.0	53%
2048	4	141.5	55%

Table 1: Performance evolution of the ParCeL-1 N-Bodies program on a T3D multicomputer with 256 PEs when the problem size grows

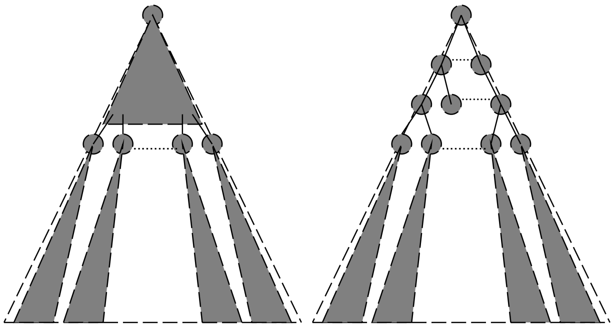

Figure 8: Two ParCeL-1 implementations of the N-queens problem: sequential and parallel splitting

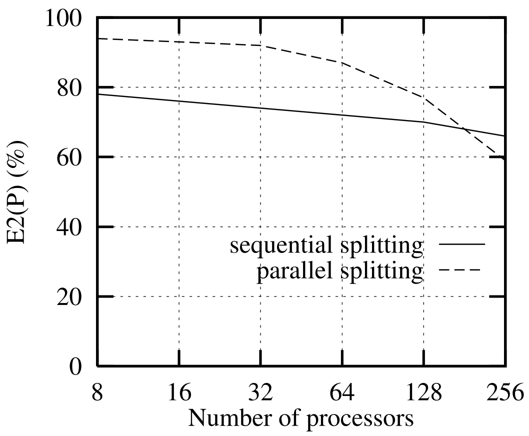

Figure 9: Efficiencies of the two 15-queens implementations on a T3D multicomputer

For 512 bodies, the performance of the ParCeL-1 program is very satisfactory. The speed-up versus a sequential C program ($SU2(P)$) and the corresponding efficiency ($E2(P)$) are presented on figures 6 and 7. The sequential slow-down $SD0$ amounts to 1.4, and limits the efficiency to approximately 70%, as shown on figure 7, even for a small number of processors. The efficiency decay for a large number of processors is due to load imbalance, communication overheads and sequential fraction problems (Amdahl's law [2]) that become critical beyond 100 processors. But performance increases with the problem size, as shown on table 1.

5.3 N-queens problem

The well known N-queens problem lies in determining how N queens can be placed on an $N \times N$ chess board so that no queen may attack an other. If we restrict the problem to the research of the number of solutions, then we have to program an exhaustive tree exploration, and each subtree can be explored independently of the others. So this problem is well-adapted to control parallelism with medium grain. We just have to split the initial tree exploration into numerous subtree explorations, to execute those in parallel, and to sum up the partial results from all subtrees. The performance is very good and better than the one of the N-bodies problem. We obtained a speed-up of up to 168 versus a sequential C program ($SU2$), on 256 processors of a T3D, for 15 queens.

We successively programmed two implementations of this problem, both splitting and pruning dynamically the search tree. The first implementation realizes the splitting sequentially, from a single cell, while the second does it in parallel, using a binary tree of cells. These two strategies are depicted on figure 8, and their respective efficiencies are summarized on figure 9. The sequential splitting program has a more important sequential slow-down ($SD0$) than the parallel splitting program, whose sequential parts have been optimized. It is therefore not as efficient as the parallel splitting version for a small number of processors. However, it scales better than the more sophisticated implementation and exhibits a better performance on 256 processors. This phenomenon may be explained by the fact that the parallel split uses more cells and channels, and therefore spends

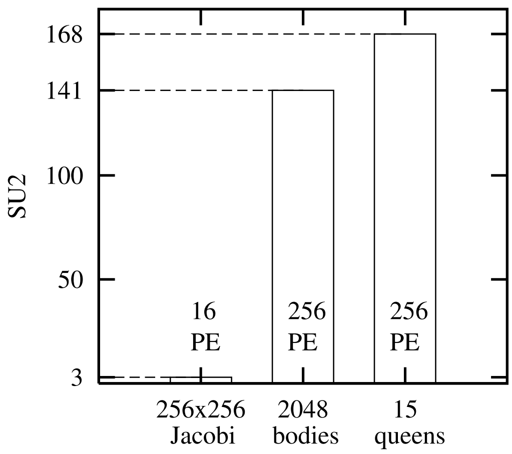

Figure 10: Best speed-ups (SU2) obtained with several test programs on a T3D multicomputer

more time to manage them. Load balancing is also more difficult for this version, because the dynamic exploration and pruning creates a great number of cells irregularly over several cycles. These problems become critical for a large number of processors.

6 Conclusion

At the time of writing, ParCeL-1 has been implemented on four parallel architectures: Telmat T-Node, Intel Paragon, Cray T3D, and a network of workstations with PVM. A same ParCeL-1 source code can be compiled and executed on any of these architectures, for any number of available processors. Moreover, ParCeL-1 hides to the user the low-level specificities of parallel programming on the different architectures.

We have tested ParCeL-1 with various benchmark programs and obtained fair to very good results. The best ones for each program are summarized on figure 10, showing that the computational model is not restricted to a specific kind of applications. We are currently working on further optimizations of the compiler, especially regarding code generation, I/O semantics and cell communications. We are also currently implementing real size applications (connectionism and artificial intelligence [11]), with large parallel I/O and data sets.

ParCeL-1 Home Page: http://www.ese-metz.fr/info/project/parcel/parcel.html

References

[1] G. Agha. *ACTORS, a model of concurrent computation in distributed systems.* MIT Press, 1986.

[2] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS conference proceedings*, Atlantic City, N.J., 1967.

[3] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant. Bulk synchronous parallel computing — a paradigm for transportable software. In *Proceedings of the 28th Annual Hawaii Conference on System Sciences*, volume II. IEEE Computer Society Press, January 1995.

[4] A. Chien, U. Reddy, J. Plevyak, and J. Dolby. IC++: A C++ dialect for high performance parallel computing. In *2nd international symposium on object technologies for advanced software (ISOTAS)*, Kanazawa, Japan, March 1996.

[5] R. Chin and S. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1), March 1991.

[6] M. W. Goudreau, H. M. D. Hill, B. McColl, S. B. Rao, D. C. Stefanescu, T. Suel, and T. Tsantilas. A proposal for the BSP world-wide standard library (preliminary version). BSP Worldwide document, April 1996. Available at http://www.bsp-worldwide.org/.

[7] M. W. Goudreau, S. B. Rao, and T. Tsantilas. A study of the BSP model: Algorithms and implementation. Technical Report UCFCS:CS-TR-94-04, Department of Computer Science, University of Central Florida, April 1994.

[8] A. S. Grimshaw, J. B. Weissman, and W. T. Strayer. Portable run-time support for dynamic object-oriented parallel processing. *ACM Transactions on Computer Systems*, 14(2), May 1996.

[9] C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI*, 1973.

[10] C. Houck and G. Agha. Hal: A high-level actor language and its distributed implementation. In *21st International conference on parallel processing (ICCPP'92)*, volume 2, St. Charles, IL, August 1992.

[11] Y. Lallement, Th. Cornu, and S. Vialle. Application development under parcel-1. In H. Kitano, C. Suttner, and J. Geller, editors, *Parallel Processing for Artificial Intelligence, 3*. Elsevier, 1996. In Press.

[12] R. Miller. A library for bulk-synchronous parallel programming. In *British Computer Society Parallel Processing Specialist Group workshop on General Purpose Parallel Computing*, December 1993.

[13] K. Taura, S. Matsuoka, and A. Yonezawa. ABCL/f: A future-based polymorphic typed concurrent object-oriented language – its design and implementation. In G. Blelloch, M. Chandy, and S. Jagannathan, editors, *Proceedings of the DIMACS workshop on specification of parallel algorithms*, 1994.

[14] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), August 1990.

[15] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 18. Elsevier Science Publishers, 1990.

[16] S. Vialle, T. Cornu, and Y. Lallement. ParCeL-1, User's Guide and Reference Manual. Technical Report R-10 (Crin number 94-R-235), Supélec, établissement de Metz, France, 1994.

[17] A. Yonezawa, J. P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *OOPSLA*, 1986.