ハイパフォーマンス
コンピューティング 57－4
（1995. 8. 24）

並列オブジェクト指向言語 ABCL/*f* による並列数値計算 — 有限要素法と多体問題による評価

田浦健次朗* 米澤明憲
E-mail: {tau, yonezawa}@is.s.u-tokyo.ac.jp
東京大学大学院理学系研究科情報科学専攻
〒113 東京都文京区本郷 7-3-1

並列オブジェクト指向言語 ABCL/*f* を用いて Barnes-Hut N 体アルゴリズム (BH アルゴリズム) および共役勾配法による有限要素解法を記述し，評価を行なった．不規則な数値計算では，ABCL/*f* のような，細かい粒度でオブジェクトを動的に生成できる言語が有効であることが確かめられた．また，我々の現在の実装の性能評価を，並列計算機 AP1000 上で BH アルゴリズムを用いて行なった．BH アルゴリズムの核部分では，現状の処理系で性能は逐次の C の約 1/3 程度であり，自明な改良によってさらに 2 倍の性能が得られることがわかった．

Irregular Numerics in Concurrent Object-Oriented Language ABCL/*f*— A Case Study in FEM and Nbody

Kenjiro Taura* Akinori Yonezawa

Department of Information Science, University of Tokyo

This paper reports our experiences on programming in concurrent object-based language ABCL/*f* by Barnes-Hut Nbody algorithm (BH algorithm) and Finite Element Method with Conjugate Gradient solver. We demonstrate that ABCL/*f* or, in general languages which encourage dynamic fine-grain object allocation has advantage for irregular numeric applications. We evaluate the language design of ABCL/*f* through these applications and give performance results obtained from Nbody on AP1000 parallel computer. In Nbody kernel, the performance of the current ABCL/*f* implementation is approximately 1/3 of sequential C and would improve by a factor of two by trivial optimizations.

*日本学術振興会特別研究員 (JSPS Research Fellow),

1 Introduction

The programming environment most common today on large scale parallel computers, the majority of which are distributed memory machines, is a sequential language such as C and Fortran + message passing library. Automatic parallelization and automatic data decomposition techniques [1, 17] achieved success on regular operations on arrays, but the state-of-the-art of automatic parallelization mainly focuses on simple DO loops with affine array indices. There are every reasons why removing these assumptions make static analysis substantially difficult. Many applications adopt irregular data structures and require explicit and application-specific locality and load-balancing control [4, 16]. Automatic parallelization or automatic data distribution for such irregular problems are beyond the ability of the current compilation technologies.

An alternative for such irregular applications is programming in languages which support *primitives* for parallel programming such as dynamic thread creation, dynamic data allocation, and location-transparent data access. Languages based on concurrent objects and/or asynchronous method/procedure invocations [5, 9] are one of the most promising and practical approaches in this direction.

This paper reports our experiences on programming in concurrent object-based language ABCL/*f* by Barnes-Hut Nbody algorithm (BH algorithm) and Finite Element Method with Conjugate Gradient solver. We evaluate the language design of ABCL/*f* through these applications and give performance results obtained from Nbody on AP1000 parallel computer [13]. The performance evaluation of FEM is currently ongoing and too premature to be presented here. Due to the space restriction, descriptions are very brief and may not be detailed enough for those who are not familiar with the algorithms. We are publishing a more detailed technical report in the near future.

2 ABCL/*f*—An Overview

ABCL/*f* is a simple parallel language based on *future* and *concurrent objects*. Future is the mean to express parallelism of various granularities and a concurrent object is the unit of encapsulation, data distribution, and mutual exclusion.

2.1 Future

Future was first proposed by Halstead [9] and has been adopted in many languages with some modifications or extensions. The basic idea is to provide *asynchronous* invocation or, a way for decoupling function (or method) invocation and obtaining results. Future serves as a building block for other monolithic parallel operations like DOALL loops or parallel function calls.

In ABCL/*f*, any user-defined procedure or method can be called both synchronously and asynchronously. The programmer does not have to decide whether a method is running in parallel with other computation in the early stage of a program development.

ABCL/*f* gives the programmer the control of the location of an invocation. On each invocation, whether sequential or parallel, the processor element (PE) number on which the computation takes place can optionally be specified. An invocation is otherwise processed locally.

2.2 Concurrent Objects

A concurrent object in ABCL/*f* (and other concurrent object-oriented languages) encapsulates stateful data and serves as a building block of complex data structures.

Methods are the mean by which a process sees/modifies the internal state of an object atomically. In addition, if the object on which a method invocation is to operate (*receiver object*) happens to be located on a remote PE, the method invocation are automatically migrated to the owner PE.

Current ABCL/*f* guarantees the atomicity of a method invocation by serializing all method invocations on a single object. This "single threaded property" of an object usually helps the programmer in that he/she does not have to keep in mind which transaction on an object may take place in parallel with other computation and hence must be protected. On the other hand, it is often argued that this rule causes unintended serialization/deadlock and precludes natural description of parallel processes which share data structure constructed from concurrent objects. We observed this in FEM code, which uses a graph structure for representing a sparse matrix. The current ABCL/*f* does not provide an adequate solution to this problem and supports an *ad hoc* escape from the rule—(possibly unsafe) methods which do not perform the automatic lock/unlock. We are working on a better object model which relaxes mutual exclusion requirement and still allows safe sharing among concurrent processes.

An object is created by invoking a constructor function, which allocates an object on the local PE. Hence object creations can be done in parallel (by future call) and object locations are specifiable by the programmer (by designating the location where the constructor is invoked).

3 The Applications

3.1 Barnes-Hut Nbody

3.1.1 Algorithm

We use the terminologies used in astrophysical simulations (e.g., mass or the center of gravity). Since the naive particle-particle method has $O(N^2)$ time complexity for N particles, improvements have been developed [8, 10]. Barnes-Hut algorithm (BH algorithm) [2] is a simple yet efficient algorithm which has $O(N \log N)$ time complexity on average. The central idea is to approximate the total of 1-to-

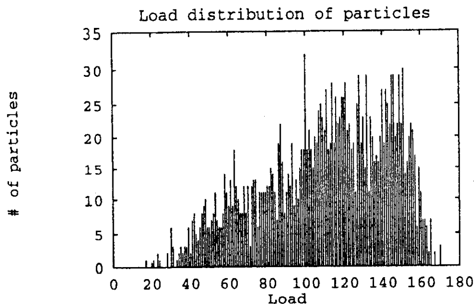

Figure 1: Load Distribution on Particles

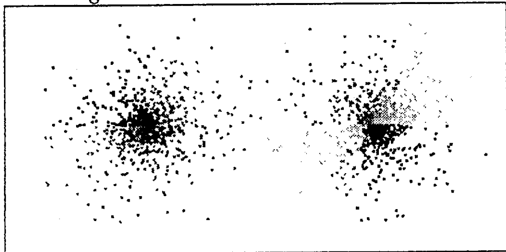

Figure 2: Example Distribution of Particles

many Newtonian forces with 1-to-1 force by regarding the "many" ones as a single particle located at the center of gravity of the particles. This approximation takes place if the distance between the two entities are far enough compared to the spatial size of the aggregate.

Before the force calculation, BH algorithm constructs a tree structure (BH tree) for the entire space which contains all the particles to be simulated. A space is divided into 2^d subspaces (where d is the dimension) if it contains more than two particles and this partition continues recursively until all leaf spaces have at most one particle. Force calculation for a particle traverses the tree until leafs or nodes far from the particle.

The irregularity of BH algorithm comes from the possibly very unbalanced tree structure, which is the consequence of irregular distribution of particles. Since the depth of the tree heavily varies depending on particle density, required amount of memory for nodes cannot be computed in advance, favoring dynamic allocation of concurrent objects. Moreover, particles in a "dense portion" perform deeper recursive calls and hence require larger amount of computation time, we cannot just simply assign an equal number of particles for each PE. Figure 1 gives the distribution of the load on each particle, the load measured by the number of Newtonian force calculations required for the particle in an example distribution of particles shown in Figure 2.

```
;;; calc-force method for node objects
;;; Call by:
;;; (calc-force node pp)
;;; pp is the position of the particle

(defmethod node calc-force (pp)
  (if contain-no-particle zero
  (if contain-one-particle
      Newtonian force between
      the particle and this node
  (if far-enough-from-the-particle
      Newtonian force between
      the particle and the center
      of gravity of this node
  (else
    call calc-force for direct children
    and sums up the results)))))
```

Figure 3: The Sequential Method to Compute a Force

3.1.2 Description in ABCL/*f*

Each particle as well as each node of a BH tree is represented by a concurrent object. Each particle object has data fields (instance variables) such as mass, acceleration, velocity, and location. Every node object keeps the rectangle it covers, the total mass of particles under the rectangle, the center of gravity of the particles. In addition, non-leaf nodes have a list of direct child nodes, whereas leaf nodes which has a particle the reference to the particle.

Here we only give description of force calculation which is the most computationally intensive phase. Other phases (tree construction, particle decomposition etc.) involve more complex, hence interesting computation than the simple kernel but cannot be given here due to the space restriction.

A typical program development begins with working on a sequential uniprocessor version and then parallelizes it. In ABCL/*f*, since programmers already use concurrent objects for representing mutable data (tree nodes and particles in Nbody case), they are usually ready to be distributed and shared by concurrent processes with small modifications. The pseudo code for the original sequential program is given in Figure 3.[1]

The most important change from the sequential version is that each PE replicates a part of the BH tree on a node-by-node basis. A node is copied when it is first accessed, in order to reduce communication traffic and, more importantly, bottleneck on nodes near the root of the BH tree. This is done by creating a "cache" object on each PE, which keeps track of association between original nodes and their local copies. The modification to the force calculation method is listed in Figure 4.

[1] A very brief explanation on the syntax:
`(defmethod ⟨class⟩ ⟨method⟩ (x y ...) ⟨body⟩)`
defines a method for the class ⟨class⟩, which are called by `(⟨method⟩ receiver x y ...)`.

```
;;; lookup cache to obtain local copy and then call
;;; the original  calc-force method

(defun calc-force-w/-cache (node pp cache)
  (let ((local-copy (lookup cache node)))
    (calc-force local-copy pp cache)))

;;; Minor change in  calc-force

(defmethod node calc-force (pp cache)
  (if ... ; first three cases are
  (if ... ; same as before
  (if ...
  ;; call calc-force-w/-cache instead
  (else
    call calc-force-w/-cache for direct
    children and sums up the results)))))
```

Figure 4: The "Parallel-Ready" calc-force

Algorithm 1

1 $x = 0$, $r = b$, $p = D^{-1}r$
2 **while** $r \neq 0$
3 $\quad \alpha \leftarrow (r,p)/(p, Ap)$
4 $\quad r \leftarrow r - \alpha Ap$
5 $\quad x \leftarrow x + \alpha p$
6 $\quad \beta \leftarrow -(D^{-1}r, Ap)/(p, Ap)$
7 $\quad p \leftarrow D^{-1}r + \beta p$
8 **end**

Figure 5: The SCG Method

3.2 Finite Element Method

3.2.1 Algorithm

Finite Element Method (FEM) is a numerical method for solving partial differential equations. We first divide the domain of the given equation into (typically triangular or rectangular) mesh. We then construct a sparse linear equation system $Ax = b$, where the dimension of A is the number of the nodes in the mesh[2]. A is a sparse matrix in that A_{ij} is zero unless node i and node j are identical or adjacent in the mesh. For example, in the regular rectangular mesh, there are at most five non-zero elements within each row.

We use the Scaled Conjugate Gradient (SCG) method for the linear solver. The algorithm in mathematical notation is given in Figure 5. In the figure, D represents the diagonal part of A.

The irregularity of FEM comes from the sparsity and the irregularity of non-zero elements of A. The sparsity of A precludes the dense matrix representation of A which stores all elements of A in a two dimensional array whose most elements are actually zero. Since A_{ij} is non-zero only if node i and j are adjacent or identical in the mesh, an irregular mesh yields accordingly irregular distribution of non-zero elements in A, which in turn results in irregular communication pattern in the matrix-vector product in the computation of Ap (in line three of Figure 5).

[2] More precisely, the dimension of A is the number of nodes minus the number of nodes on edges where Dirichlet condition holds.

```
;;;  compute-Ap method called
;;; for all nodes in the mesh
;;; Each node computes an element
;;; of  Ap

(defmethod node compute-Ap ()
  (let ((s 0.0))
    foreach x in  neighbors-list
      (setq s (+ s (* (get-p (node  x))
                      (coeff  x))))
    (setq Ap s)))
```

Figure 6: The Method which Computes Ap

3.2.2 Description in ABCL/f

We define a *node* object which corresponds to a row in the equation. For each vector which appears in the solver (i.e., b, x, r, and p), a node object has a corresponding instance variable which stores the current value of the element in the row. A node object also has a list of adjacent nodes in the mesh and the corresponding possibly non-zero elements in A, and information such as the location of the node, which are not relevant to the solver but necessary for displaying purpose etc.

The solver has two communication patterns: (1) in the computation of Ap, a node requires communication with adjacent nodes to get the value of p, (2) in the computation of α or β, a global summation is performed for the inner products.

Figure 6 shows a method used in the solver which compute Ap in the third line of Figure 5). It is much like a sparse matrix vector product using index array.

The mesh we used as an example is a triangular mesh, which is somewhat more irregular than the rectangular mesh, but still far simpler than those which appear in real engineering code. This is for the simplicity of our mesh generation code and data decomposition. Our solver by no means relies on any property of the particular mesh we selected here.

Our FEM program is still in very early stage of development and it has already turned out that some sort of scheme must be used for reducing communication overhead and latency. We are investigating how optimizations such as message vectorization or prefetching are expressed without destroying the structure of the original loop. Replication technique used in Nbody will be effective here too.

4 Performance Evaluation

4.1 Evaluation Strategy

Evaluation of a *language implementation* is not a trivial task. The difficulty lies in the fact that while we are interested in the cost of *fundamental* operations such as method invocation or the overhead of sequential computation, simple synthesized benchmarks do not give us really interesting information. We would like to somehow isolate the cost of primitives *within* a reasonably complex application.

In the force-calculation of BH algorithm, each PE performs a tree traversal, replicating remote nodes

```
/* m: mass, P, Q: location */
vect_t newton_acceleration (m, P, Q)
{
    float dx = P->x - Q->x;
    float dy = P->y - Q->y;
    float rr = dx * dx + dy * dy;
    float r = sqrt (rr);
    float c = mass / (r * rr);
    return make_vect (c * dx, c * dy);
}
```

Figure 7: The Sequential Kernel Written in C

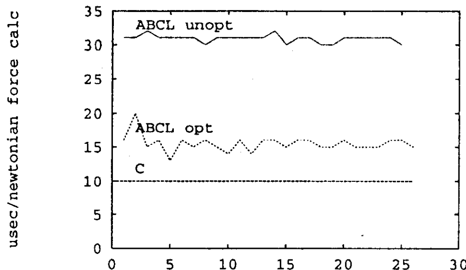

Figure 8: The Performance of the Nbody Kernel

when they are first accessed by the PE. For the simplicity, each PE does not issue multiple replication requests in parallel. These facts enable us to isolate the cost of two fundamental operations, which are (1) remote replication and (2) Newtonian force calculation (i.e., $G\frac{mM}{r^3}\vec{r}$).[3] The former is determined by artificially traversing the BH tree twice; since nodes required by each PE has already been replicated in the first traversal, the second traversal requires no communication. Hence the difference between these two phases tells us how much time are spent on communication.[4] Similarly, the later can be computed by replacing Newtonian force calculations at leaves by a trivial operation and comparing with the original running time.

A subtly arises because many PEs are running in parallel with possibly imbalanced loads. When we say "running time," it effectively refers to the running time of the most heavily loaded PE. Therefore, we first perform a profiling run to determine the most heavily loaded PE and then examine the difference on that PE. The profiling also tells us the number of operations performed, hereby we can determine the cost per operation. Throughout the evaluation, we exclude the cost of garbage collection, which currently incurs a disruptive pause time. We will be evaluating them in the future with a concurrent garbage collector [11].

Figure 8 shows the cost (in μseconds) of a Newtonian force calculation in ABCL/f and a synthesized

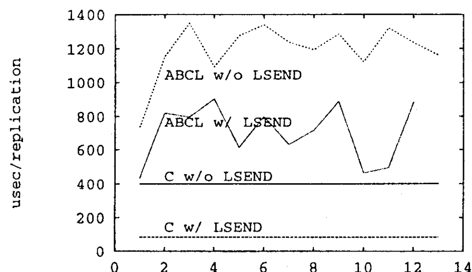

Figure 9: The Performance of Communication

sequential C program. The C version repeatedly calls a C function shown in Figure 7. Both programs run on AP1000, a node CPU of which is a 25MHz Sparc. We used single floating point numbers for saving space and compiled with GNU C compiler version 2.6.3 which emits single float instructions. From the figure, we observe that the implementation is slower than C by a factor of three. A thorough look at the generated code reveals that significant improvement will be possible by inlining calls to `sqrt` and `make_vect`. In ABCL/f, `sqrt` is a function that may be called by future, which then calls the standard `sqrt` *à la* `libc`. Our current implementation takes about ten instructions to perform an unknown call and perform a polling for each unknown call. Unfolding these two calls and eliminating pollings, which is by no means a difficult optimization, immediately gets 50% speedup, as indicated in the center line in Figure 8.

Figure 9 compares communication performance of ABCL/f in Nbody and a synthesized C benchmark, which repeatedly makes round trips of one word messages. AP1000 communication library supports two ways for message passing and we tried both options.[5] A replication involves a pair of request/reply messages and runtime data parsing which flattens/unflattens the data to be sent into/from a contiguous memory block. The current implementation does not put efforts on communication performance tuning, favoring the easiness of porting to other platforms. It first flattens the message using runtime tag into a message buffer and then call a standard bulk transfer library. The receiver side interprets runtime tags in the message and expand the data structure into the heap. The performance result shown in Figure 9 indicates that a significant time is spent on the runtime data structure parsing.

5 Related Work

Nbody and FEM have been drawing so great scientific/engineering interest that many researchers

[3] In the actual program, we set G as 1.0 and do not multiply m since it is canceled after all.

[4] Here we ignore the variation of cache lookup time.

[5] LSEND is a faster communication library at the risk of message buffer overflow.

have implemented efficient code on commercial distributed memory MPPs [3, 7, 14, 15]. As far as we know, the programming environment they used on distributed memory machines has been C or Fortran + message library. In such programming environment, significant programming efforts have been put on managing communication and contexts blocked until the result of a communication arrives.

A controversial language design decision in ABCL/f is that, unlike many other concurrent object-oriented or Lisp-based languages [5, 9], ABCL/f lets the programmer specify the data/computation location. Although specifying data/computation location is troublesome task, there must be some way by which programmer controls them, given that communication on scalable parallel processors is an expensive thing. Design decision made in ABCL/f is of course unexciting, but useful for intensive parallel applications and serves as the base language on top of which more location-transparent layers can be defined.

A good alternative is to support shared name space, while giving the programmer the control over coherence semantics and consistency management protocols. There have been previous works [6, 12], but they are mainly targeting LAN environment where the significant messaging overhead favors constant software consistency management overhead. It has not been proved that this gives the programmer higher level of location transparency, without sacrificing local computation on MPPs context.

6 Conclusion

Descriptions of two representative irregular scientific applications in concurrent object-oriented language ABCL/f, experimental result of Nbody, and the evaluation of the current language implementation were given.

Our future research direction includes more detailed analysis of Nbody and other applications and improving the implementation. We are also developing better object-model which allow more sharing, migration, and replication with small overhead. Establishing simple, machine-independent (working whether the machine is DM or DSM machine), performance transparent, and efficiently implementable (again both on DM and DSM) object model will be the valuable work to pursuit.

References

[1] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proc. of '93 ACM SIGPLAN Programming Language Design and Implementation*, pages 112–125, July 1993.

[2] Josh Barnes and Piet Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[3] Graham Carey, Joe Schmidt, Vineet Singh, and Dennis Yelton. A scalable, object-oriented finite element solver for partial differential equations on multicomputers. In *Proceedings of 1992 International Conference on Supercomputing*, pages 387–396, 1992.

[4] Andrew Chien, M. Straka, Julian Dolby, Vijay Karamcheti, John Plevyak, and Xingbin Zhang. A case study in irregular parallel programming. In *Proceedings of the DIMACS workshop on Specification of Parallel Algorithms*, 1994.

[5] Andrew A. Chien. *Concurrent Aggregates (CA)*. PhD thesis, MIT, 1991.

[6] Tzi-cker Chiueh and Manish Verma. A compiler-directed distributed shared memory system. In *Proceedings of International Conference on Supercomputing*, pages 77–86, 1995.

[7] Ananth Y. Grama, Vipin Kumar, and Ahmed Sameh. Scalable parallel formulation of the Barnes-Hut method for n-body simulations. In *Proceedings of Supercomputing '94*, pages 439–448, 1994.

[8] Leslie Greengard. *The Rapid Evaluation of Potential Fields in Particle Systems.* ACM Distinguished Dessertations. The MIT Press, 198.

[9] Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[10] R. W. Hockney and J. W. Eastwood. *Computer Simulation using Particles.* Institute of Physics Publishing, 1988.

[11] Tomio Kamada, Satoshi Matsuoka, and Akinori Yonezawa. Efficient parallel global garbage collection on massively parallel computers. In *Proceedings of Supercomputing '94*, pages 79–88, 1994.

[12] Steven K. Reinhardt, James R. Larus, and David A. Wood. Tempest and typhoon: User-level shared memory. In *Proceedings of International Conference on Computer Architecture*, pages 325–336, 1994.

[13] Toshiyuki Shimizu, Takeshi Horie, and Hiroaki Ishihata. Low-latency message communication support for the AP1000. In *The 19th Annual International Symposium on Computer Architecture*, pages 288–297, 1992.

[14] Michael S. Warren and John K. Salmon. Astrophysical N-body simulations using hierarchical tree data structures. In *Proceedings of Supercomputing '92*, pages 570–576, 1992.

[15] Michael S. Warren and John K. Salmon. A parallel hashed oct-tree N-body algorithm. In *Proceedings of Supercomputing '93*, pages 12–21, 1993.

[16] A. J. Wing. *Discrete Event Simulation in Parallel*, chapter 7, pages 179–226. Blackwell Scientific Publications, 1992.

[17] Hans Zima. *Super Compilers for Parallel and Vector Computers.* Frontier Series. ACM Press, 1991.