Data Allocation for Parallel Processing in Distributed Computing Systems

Alexey Syschikov
Saint-Petersburg State University
of Aerospace Instrumentation
190000, Bolshaya morskaya str. 67,
Saint-Petersburg, Russia
alexey.syschikov@guap.ru

Denis Rutkov
Saint-Petersburg State University
of Aerospace Instrumentation
190000, Bolshaya morskaya str. 67,
Saint-Petersburg, Russia
denis.rutkov@guap.ru

Abstract

Efficient parallelization of equal operations of data array elements processing is not a new problem. This is video processing, matrix computations, scientific calculations, cryptanalysis and much more – all these calculations have great potential and abilities opportunities for parallelization. However, the problem of finding high-quality solution for this task is still actual.

The subject of this article will be systems with massive parallelism and distributed memory. For this class of parallel systems, this problem is solving with varying success for over 30 years.

An analysis of the current situation revealed that for the effective application of a programming language the very important question is a description of allocation of source data to local memory of processor elements (PEs) in distributed computing system. It occurs every time when new programming language or compilation system is been developed. Over the years there where developed a lot of not just implementations, but even approaches for solving of this task.

In this article we'd like to make an analysis of existing approaches to the data allocation, consider their pros and cons, the activity of their application in programming languages and systems.

Index Terms: parallel programming, data distribution, massive parallelism, distributed systems.

I. Introduction

Efficient parallelization of equal operations of data array elements processing is not a new problem. This is video processing, matrix computations, scientific calculations, cryptanalysis and much more – all these calculations have great potential and abilities opportunities for parallelization. However, the problem of finding high-quality solution for this task is still actual.

Of course, for some classes of computer systems it is solved sufficiently. Vector computation systems are specially designed to solve such problems, although such computations have very strict limitations [1] (no nested cycles, no branching, no procedures and functions calls, no recursion, etc.). With sufficient effectiveness this problem is solving in parallel systems with shared memory (performance is provided by the architecture of the system) both with traditional methods (threads, fibers, etc.) and with the help of parallelizing compilers. At the same time there left fewer restrictions than on vectorization.

The subject of this article will be systems with massive parallelism and distributed memory. For this class of parallel systems, this problem is solving with varying success for over 30 years.

An analysis of the current situation revealed that for the effective application of a programming language the very important question is a description of allocation of source

data to local memory of processor elements (PEs) in distributed computing system. It occurs every time when new programming language or compilation system is been developed. Over the years there where developed a lot of not just implementations, but even approaches for solving of this task.

In this article we'd like to make an analysis of existing approaches to the data allocation, consider their pros and cons, the activity of their application in programming languages and systems.

II. MAIN PART

A. Concept of data allocation

There are two key aspects in the data allocation task. The first aspect is a type of allocation (how data are allocated on PE's of a distributed system), and the second aspect is methods of allocation description (what language instruments are presented for programmer to define and control data allocation).

1 Allocation types

Two main types of data allocation to processing elements in distributed data processing can be specified: data localization and data distribution.

1.1 Data distribution

PEs receive just a part of data that is required for them to perform calculations, for the remaining data it is necessary to request data allocated on other PEs.

Pros:

- Ability to organize parallel computations with complex structure of data dependences (currently we exclude the question of their effectiveness).
- Ability to process data without increasing amount of data in system or increase it only in order to optimize computations.

Cons:

- Reducing computation effectiveness due to waiting for communication results with other PEs.
- The necessity for synchronization of inter-processor communications and taking into account data communications when writing program code.

1.2 Data localization

To perform an allocated part of computations PE receives sufficient for them set of data and no additional data will be required.

It is clear that data localization is a particular case of the data distribution, but it looks reasonable to consider this type as a separate one. At first, the fundamental difference is that there are no communications while local computations are performed. The second is that some formal computational models can not access distributed data t all, for example CSP-model [2] of Charles Hoare and derived from it, Data-Flow model [3] and some others.

Pros:

- PEs are completely independent from their environment when doing computations, which makes easier both organization of calculations and the system organization in a whole.

- No communications in the computation process, that allows PE to perform computations without delays concerned with data exchange and synchronization.

Cons:

- Increasing number of data in the system due to duplicating data for localization that increases usage of computation system communications.
- The requirement to centralization and preparing of data for further localization and to collect results of this data processing.
- Impossibility to parallelize computations that have non-localizable data dependencies.

2 Allocation specification methods

2.1 Automatic allocation

Automatic allocation implies that data in parallel computation will be allocated among PEs without the involvement of the programmer. Thus the data allocation task must be fully solved by a compiler, which usually calculates the allocation through the analysis of dependencies. On the current moment the automatic data allocation is mainly implemented in compilers with automatic parallelization, although formally it can be combined with an explicit description of parallelism.

2.2 Predefined allocation

Predefined allocation implies that the programmer is given a limited set of allocation templates. In specification of parallel computations data will be allocated on the set of virtual processors according to the selected template. The mapping of virtual processors to physical processors is implementation-dependent and is performed by a compiler or in runtime.

Templates of allocations are usually parameterized, for example, programmer can specify the division of the array into parts with certain step, or for certain dimensions.

2.3 Custom allocations

Custom allocations should be completely specified by a programmer. Besides that they may have both a template form that can be applied for several allocations and the form of explicit splitting of the source data array using some operators. In the second case, the programmer often needs also to program the communications to allocate the split data sets.

As it was for predefined allocations when using custom allocations a programmer also allocate data to virtual processors, i.e. it is a form of hints to a compiler or a runtime environment for best organization of independent parallel computations.

B. Application of data allocation concepts in programming languages

In the context of the specified theme we will examine in details what approaches were used or currently are used in different programming languages and systems for distributed systems.

From the examined list of languages and systems we exclude ones that are oriented on obvious parallelism using processes and ones that uses only direct message passing “point-to-point” (Ada, Erlang, etc.). Such approach to data allocation gives nearly unrestricted abilities of allocation; however allocation implementation fully falls on the programmer shoulders. It dramatically complicates the program code and may lead to significant decreasing of computation effectiveness if implemented inaccurately.

Within article format we can not include detailed examples of all described mechanisms, but we shall try to illustrate all basic concepts.

1 MPI (Message Passing Interface)

MPI is a message processing library (C/C++ functions or Fortran subroutines) which provides communication routines between single processes of parallel program in the systems with distributed memory (both for data exchange and tasks synchronization). Currently MPI is commonly accepted as a standard and is the most developed library of parallel programming with message passing. [4, 5, 6].

1.1 Allocation types

MPI 1.x standard defines data localization.

MPI 2.x standard defines extended facilities of data distribution by direct read/write access to the allocated memory of specified remote process

1.2 Allocation specification methods

MPI functions organize predefined allocations with blocks. Functions SEND, SCATTER, GATHER etc.

– Equal blocks. Parameter – amount of data for block.

```
MPI_Scatter( sendbuf, 15, MPI_INT, rbuf, 15, MPI_INT, root, comm);
// Where 15 – a block size for every destination process
```

/* Process root split source data (stored at address sendbuf) between all processes of communicator comm. with equal blocks of 15 integers for every process. */

– Unequal blocks. Parameters – two arrays that contains pointers to data for each process and amount of data for each process respectively.

```
MPI_Comm comm;
int gsize,sbuf[1000];
int displ, rbuf[1000], i, disp[1000], cnt[1000];
...
MPI_Comm_size( comm, &gsize);
displ = 0;
for (i=0; i<gsize; ++i) {
   disp[i] = displ;
   cnt[i] = i*2;
   displ += cnt[i];
}
MPI_Scatterv( sbuf, cnt, disp, MPI_INT, rbuf, cnt[nproc], MPI_INT,
0, comm);
```

/* Where disp – array with displacements in the source array, cnt – array with block sizes

Process root split source data (stored at address sbuf) between all processes of communicator comm. with different blocks of 0, 2, 4, etc. integers for processes 0, 1, 2, etc. respectively. */

Unequal blocks data allocation mechanism allows to make a data duplication thus, in a common, this data allocation may be considered as the custom allocation.

2 HPF (High Performance Fortran)

HPF programming language aimed to reach the following goals [7]:

- Support programming for scaling parallel systems (especially parallelization by data model)
- Provide hardware independent programming model with 3 main features:
  - Programmers should be able to consider memory as a global address space, even on distributed memory systems. In other words, arrays should be globally accessible but locally distributed over memory of PEs that take place in a computation.
  - There should be look like as if there is a single control flow, so that a program could be run on a single-processor system; all parallelism should arise from parallel application of operations on distributed data (SPMD model).
  - Communication operations code should be generated implicitly, so that the programmer won't need to define and control PEs interaction.
- Allow code generation with the same level of performance as in a manually tuned MPI-based implementation.

In the process of the language standard developing it was assumed that modern (for that time) compilers cannot both automatically and efficiently allocate data between PEs. To solve this problem special directives were introduces to HPF standard which allow specifying data allocation. In HPF data allocation is done in several steps:

- Object alignment. Directives ALIGN, REALIGN.
- Data allocation between virtual processor mesh. Directive DISTRIBUTE.
- Mapping virtual processors to real processors (is implementation-dependent and is performed by a compiler).

2.1 Allocation types

HPF uses data localization. The programming language doesn't allow defining explicit inter-processor communications or access data elements located on other PE.

2.2 Allocation specification methods

In HPF language (version 1.x) there are used predefined distributions of equally-sized blocks (BLOCK and CYCLIC directives). They can take data amount for each PE as a parameter [8].

```
REAL VECT(10000)
!HPF$ DISTRIBUTE VECT(BLOCK)
```

!Specifies that the array VECT should be distributed across some set of abstract processors by slicing it uniformly into blocks of contiguous elements. If there are 50 processors, the directive implies that the array should be divided into groups of 200 elements, with VECT (1:200) mapped to the first processor, VECT (201:400) mapped to the second processor, and so on.

```
INTEGER GO_BOARD(19,19)
!HPF$ DISTRIBUTE GO_BOARD(CYCLIC,*)
```

!The CHESS_BOARD array will be carved up into contiguous rectangular patches, which will be distributed onto a two-dimensional arrangement of abstract processors. The GO_BOARD array will have its rows distributed cyclically over a one-dimensional arrangement of abstract processors. (The "*" specifies that GO_BOARD is not to be distributed along its second axis; thus an entire row is to be distributed as one object. This is sometimes called "on-processor" distribution.)

Language standard also includes a set of functions, which perform global array operations (sum, reduction etc, prefix-based operations). From a programmers point of view such operations looks like operations with automatic distribution. However we assume that they are implemented with standard predefined distributions.

```
SUM_SCATTER((/1, 2, 3, 1/), (/4, -5, 7/), (/1, 1, 2, 2/))
```

!Scatters elements of ARRAY (/1, 2, 3, 1/) selected by MASK (/1, 1, 2, 2/). Each element of the result is equal to the sum of the corresponding element of BASE (/4, -5, 7/) and the elements of ARRAY scattered to that position. The result will be [7 -1 7].

HPF 2.0 standard includes optional extensions, which add control over data distribution (GEN BLOCK and INDIRECT), as well as distribution modifiers (RANGE and SHADOW). [9]

3 Co-array Fortran (Fortran 2008)

In spite of active programming of tasks with high computation complexity, the parallel processing has no any tools in the Fortran programming language up to the current time (version 2003). It leads to appearance of a set of unofficial parallel extensions to the base standard. Only in prospective standard Fortran 2008 it is planed to include parallel abilities.

On the current moment the ISO Fortran Committee decided to include into the next Fortran standard nearly "as is" the unofficial Fortran extension named Co-Array Fortran or F—. Thus it looks reasonable to examine abilities of this extension as abilities of the Fortran language.

Co-Array Fortran is a small set of extensions to Fortran 95 for Single Program Multiple Data, SPMD, parallel processing. It is a simple, explicit notation for data decomposition, such as that often used in message-passing models, expressed in a natural Fortran-like syntax. The syntax is architecture-independent and can be implemented either on distributed memory machines or on shared memory machines or on clustered machines etc. [10, 11, 12]

3.1 Allocation types

Co-array Fortran language uses data distribution. In fact, Co-array Fortran is a high-level superstructure over traditional communication routines such as MPI, for example.

3.2 Allocation specification methods

In the Co-array Fortran language it is used the predefined allocation with fixed-size blocks. An array is extended with external dimensions which are allocated to virtual processors. Exact execution stream should obviously access either to its local copy of an array or to a remote one.

```
X     = Y[PE]     ! get from Y[PE]
Y[PE] = X         ! put into Y[PE]
Y[:]  = X         ! broadcast X
Y[L]  = X         ! broadcast X over subset of PE's in array L
Z(:)  = Y[:]      ! collect all Y
```

4 DVM (Distributed Virtual Machine, Distributed Virtual Memory)

DVM-system allows developing parallel programs in C-DVM and Fortran-DVM languages for different architecture computers and computer networks. The DVM name originates from two notions - Distributed Virtual Memory and Distributed Virtual Machine. The former reflects the global address space, and the latter reflects the use of virtual machines for the two-step mapping of data and computations onto a real parallel computer. [13, 14]

Using C-DVM and Fortran-DVM languages a programmer deals with the only version of the program both for sequential and parallel execution. Besides algorithm description by means of usual C and Fortran 77 features the program contains rules for parallel execution of the algorithm. These rules are syntactically organized in such a manner that they are "transparent" for standard C and Fortran compilers and doesn't prevent DVM-program execution and debugging on workstations as usual sequential program.

4.1 Allocation types

DVM-system uses both types of data allocation:

- Data localization (directive DISTRIBUTE).
- Data distribution (directives SHADOW, REMOTE, REDUCTION). There are two kinds of specifications: synchronous and asynchronous for all types of remote references. Synchronous specification defines group processing of all remote references for given statement or loop. During this processing, requiring communications, execution of the statement or the loop is suspended. Asynchronous specification allows overlapping computations and communications. It unites remote references of several statements and loops. To start reference processing operation and wait for its completion, special directives are used. Between these directives other computations, that don't contain references to specified variables, can be performed.

4.2 Allocation specification methods

DVM-system uses predefined allocations:

- Equal blocks (localization / distribution). Parameter – amount of data for block or for equal blocks between all processors.

```
CDVM$  PROCESSORS R( 4 )
REAL A(12)
CDVM$  DISTRIBUTE A (BLOCK) ONTO R
```

!Split the array A between amount of processors specified in R with equal blocks of |A|/|R| elements. For some values of |A| and |R| last processors in R may receive less array elements or even receive nothing.

– Unequal blocks (localization). Parameter – array with data amount for every processor or array with weights of source data elements.

```
CDVM$  PROCESSORS R( 4 )
INTEGER BS(4)
REAL A(12)
CDVM$   DISTRIBUTE A ( GEN_BLOCK( BS ) ) ONTO R
```

!Split the array A on |BS| blocks, where block i has size BS(i) and is placed on processor R(i),

– Allocation over alignment.

```
REAL A(10), B(10,10), C(10)
CDVM$  DISTRIBUTE B ( BLOCK , BLOCK )

CDVM$  ALIGN A( I )      WITH B( 1, I )
```

!Alignment on array section (vector alignment over the first row of matrix A)

```
CDVM$  ALIGN C( I )      WITH B( *, I )
```

!Vector multiplication (alignment of vector over every rows of matrix B)

5 ZPL

ZPL is an array programming language. Although it doesn't contain explicit parallelizing instructions, it uses the array abstraction to implement a data parallel programming model. Data is automatically distributed based upon logic of computation specified by the programmer [15, 16].

Arrays are addressed in the computation as a whole variable; indexation is specified separately with the concept of regions. A region is a set of integers bounded by number of dimensions and inner bounds. A two-dimensional region bounds a rectangular area in a two-dimensional array.

Array names, regions and operators define a computational pattern similar to cycles in other languages. Without additional specifications it is assumed that all arrays within single computation are indexed symmetrically.

The programmer can specify array and region shifts in one or multiple dimensions. Shifts could be used to describe dependencies between cycle iterations. In more traditional languages cycles would be using shifted indices when accessing arrays.

ZPL also supports special data-oriented operators: reduction, flood, gather and scatter. These operators also support regions, which determine data access patterns. For example, matrix row sum or column replication can be expressed through these operators.

5.1 Allocation types

ZPL execution model uses data distribution.

Shifted arrays will probably lead to data interchange between PEs because of inter-computational dependencies and therefore bring delays to the execution. Reduce and flood operators will lead to even greater communication volume.

This is described in the ZPL WYSIWYG model. The programmer, even without knowledge about data allocation in the system, can estimate performance of the program operations.

5.2 Allocation specification methods

ZPL execution model supports automatic allocation only. And only block allocation is implemented which equally partition the array between virtual PEs. Arrays which are used in single computation are automatically aligned, so that co-used array elements will get to the same PE

```
---------- Declarations ----------
region
  R    = [1..n,   1..n  ]; -- problem region
  BigR = [0..n+1, 0..n+1]; -- with borders
direction
  north = [-1,  0]; -- cardinal directions
  east  = [ 0,  1];
  south = [ 1,  0];
  west  = [ 0, -1];
---------- Entry Procedure ----------
procedure jacobi();
var
  A, Temp : [BigR] float;
  delta   :        float;
[R] begin
  repeat
    Temp := (A@north + A@east + A@south + A@west) / 4.0;
    delta := max<< abs(A-Temp);
    A := Temp;
  until delta < epsilon;
end;
```

--Jacobi iteration. Given an array A, iteratively replace its elements with the average of their four nearest neighbours, until the largest change between two consecutive iterations is less than epsilon.

6 Chapel

Chapel is a new high-performance programming language supporting OOP, currently being developed by Cray company. Chapel was designed from first principles rather than by extending an existing language. It is an imperative block-structured language, designed to be easy to learn for users of C, C++, Fortran etc. While Chapel builds on concepts and syntax from many previous languages, its parallel features are most directly influenced by ZPL, High-Performance Fortran (HPF), and the Cray MTA's extensions to C and Fortran.

From ZPL Chapel inherited and extended region concept renaming them as domains. Array operations are described through forall operator using corresponding domains and iterators. Iterators are used to address array elements inside the forall cycle. [17, 18].

6.1 Allocation types

Chapel uses data distribution. Arrays and variables are mapped to several memory regions called locales. A locale can be viewed as a memory local to a virtual PE. Accessing data in another locale can lead to inter-processor communication and to execution delays.

6.2 Allocation specification methods

Chapel is stated to be oriented towards custom allocations. However, it will also support a set of common pre-defined distributions like Block, Cyclic, BlockCyclic, Cut.

To define a new distribution the programmer creates a new class inherited from the built-in Distribution class. In this new class a map function must be implemented, which takes an array index and returns a corresponding locale [19].

Multidimensional distributions and locale sets (i.e. processor meshes) are supported.

```
const n1 = 1000000;
class MyC: Distribution {
  const z: integer; /* block size */
  const ntl: integer =... /* number of target locales */
function map(i:index(source)): locale { return
Locales(mod(ceil(i/z-1)+1,ntl));}
}/*Global map for a simplified block-cyclic distribution with block size z≥1; the type of argument i is the type of the indices in the source domain: */
class MyB: Distribution {
const bl: integer =. . .; /* block length */
function map(i: index(source)): locale { return
Locales(ceil(i/bl));}
}
   /*Global map for a simplified regular block distribution with block length bl: */
const D1C: domain (1) distributed (MyC(z=100))=[1..n1];
const D1B: domain (1) distributed (MyB)
   on Locales(1..num locales/10)=[1..n1];
var A1: [D1C] float;
var A2: [D1B] float;
```

Instead of creating a new distribution the programmer can use alignments. Alignments allow defining a distribution relative to another distribution.

Apart from global array allocation the programmer can define allocation of array elements inside the locale. For example, this can be useful in implementing sparse matrices.

```
   // Sparse domain declaration
const D: domain(2) = [1..n, 1..n];
var SpsD: sparse subdomain(D);

   // Sparse domain assignment
SpsD = ((1,1), (n,n));
SpsD = [i in 1..n] (i,i);
SpsD = readIndicesFromFile("inds.dat");
   // Sparse domain modification
D += i;
D.add(i);
D -= i;
D.remove(i);
```

III. CONCLUSION

A. Comparison table of presented languages

Language or system	Allocation types		Allocation specification		
	Localization	Distribution	Automatic	Predefined	Custom
MPI	+	+ (2.x)	–	+	~
HPF	+	–	~	+	–
Co-array Fortran	–	+	–	+	–
DVM	+	+	–	+	–
ZPL	–	+	+	–	–
Chapel	–	+	–	+	+

B. Summary

Old problem of finding of compromise between programmability and functionality also appeared around the data allocation in distributed systems problem.

From the above table it can be seen that most approaches use the predefined allocations. In most cases there are just several predefined allocations: about 5 templates. And this can be easily understood: the small amount of fixed templates allows adding them into the language syntax and effectively implement into the compiler.

At the same time, a limited amount of allocation templates significantly restrict abilities of a programmer to implement his task conveniently. This is especially important for sparse data and adaptive algorithms. In this case an absence of allocation control ability may force programmer to stop programming on this language. (For example, presence just of three predefined allocations in HPF was one of the reasons of loss of interest to this standard [7]). Otherwise he should implement preliminary data transformation to be able to use allocation template. Often it leads to significant loss in program effectiveness both in performance and in memory requirements.

The analysis shows that selection between data localization and data distribution is always a compromise. Implementation of both methods increases the language capabilities but at the same time makes much more sophisticated its application as well as its development. As an example, the DVM-system contains extended functionality of data localization and allocation thus it requires extra time from a programmer to study the language. And even more time is required to effectively use these language abilities in product development.

The language containing small set of predefined allocations looks as reasonable compromise provided those allocations cover the majority of typical allocation tasks. Additionally the language must include constructions for description of custom allocations.

At the same time clear choice must be done between data localization and data allocation. And this decision shall be based on common properties of the tasks and computation systems on which the selected language is mainly directed to.

REFERENCES

[1] Programs vectorization: theory, methods, implementation - coll./ Edited by. G.D.Chinin, 1991

[2] C. A. R. Hoare. Communicating Sequential Processes. *Prentice Hall International Series in Computer Science*, 2004 (1985).

[3] Alan L. Davis, Robert M. Keller. Data Flow Program Graphs. *University of Utah*, 1982.

[4] Message Passing Interface Forum. http://www.mpi-forum.org/

[5] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, Jack Dongarra. MPI: The Complete Reference. *The MIT Press, Cambridge, Massachusetts, London, England*, 1996.

[6] Olenev N.N. Parallel programming in MPI interface. *Dorodnicyn Computing Centre, Russian Academy of Sciences*. http://www.ccas.ru/mmes/educat/lab04k/

[7] Ken Kennedy, Charles Koelbel, Hans Zima. The Rise and Fall of High Performance Fortran. An Historical Object Lesson, *Proceedings of the third ACM SIGPLAN conference on History of programming languages 2007, San Diego, California*, June 09 - 10, 2007.

[8] High Performance Fortran Language Specification, version 1.1. High Performance Fortran Forum, 1994. http://hpff.rice.edu/versions/hpf1/hpf-v11/hpf-report.html

[9] High Performance Fortran Language Specification, version 2.0. High Performance Fortran Forum, 1997. http://hpff.rice.edu/versions/hpf2/hpf-v20/index.html

[10] Alan Wallcraft. Co-Array Fortran. http://www.co-array.org/

[11] Co-Array Fortran at Rice University. http://www.hipersoft.rice.edu/caf/

[12] R. W. Numrich2 and J. K. Reid. Co-Array Fortran for parallel programming. *Department for Computation and Information, Rutherford Appleton Laboratory, Oxon, UK*, 2008.

[13] DVM system. *Keldysh Institute of Applied Mathematics, Russian Academy of Sciences*. http://www.keldysh.ru/dvm/index.html

[14] N. A. Konovalov, V. A. Krukov, Yu. L. Sazanov. C-DVM-A Language for the Development of Portable Parallel Programs. *Programming and Computer. Software*, 1999.

[15] A Portable, High-Performance Parallel Programming Language for Science and Engineering Computations. *University of Washington*. http://www.cs.washington.edu/research/zpl/home/index.html

[16] Bradford L. Chamberlain, Sung-Eun Choi, Steven J. Deitz, and Lawrence Snyder. The high-level parallel language ZPL improves productivity and performance. *In Proceedings of the IEEE International Workshop on Productivity and Performance in High-End Computing*, 2004.

[17] Chapel, the Cascade High-Productivity Language. *Cray Inc*. http://chapel.cray.com/

[18] Bradford L. Chamberlain, David Callahan, Hans P. Zima. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications archive*, Volume 21, Issue 3, 2007.

[19] An Approach to Data Distributions in Chapel. Roxana E. Diaconescu and Hans P. Zima. *International Journal of High Performance Computing Applications*, August 2007, 21(3): 313-335.