Parallel Programming with Interacting Processes

Peiyi Tang[1] and Yoichi Muraoka[2]

[1] Department of Mathematics and Computing
University of Southern Queensland
Toowoomba 4350 Australia
[2] School of Science and Engineering
Waseda University
Tokyo 169 Japan

Abstract. In this paper, we argue that interacting processes (IP) with multiparty interactions are an ideal model for parallel programming. The IP model with multiparty interactions was originally proposed by N. Francez and I. R. Forman [1] for distributed programming of reactive applications. We analyze the IP model and provide the new insights into it from the parallel programming perspective. We show through parallel program examples in IP that the suitability of the IP model for parallel programming lies in its programmability, high degree of parallelism and support for modular programming. We believe that IP is a good candidate for the mainstream programming model for the both parallel and distributed computing in the future.

Keywords: *Programming Models, Parallel Programming, Interacting Processes, Multiparty Interactions, Programmability, Maximum Parallelism, Modular Programming.*

1 Introduction

The concept of parallel computing for high performance has been around for decades. While the technology of hardware and architecture has allowed to build powerful scalable parallel machines like CM-5, SP/2 and AP3000, the software to run those machines remains scarce. Parallel programming has proved to be hard. The productivity of parallel software development is still low. One of the reasons for the gap between parallel software and machines is the lack of appropriate model for parallel programming.

The parallel programming models currently accepted include the explicit communication model, the distributed shared-memory model and the data-parallel model [2]. All these models try to deal with the fact that an individual processor of a scalable parallel machine cannot hold the entire memory space of large problems. The data space of a large problem has to be distributed among physical local memories of the parallel processors.

The explicit communication model uses the abstraction of direct communication to allow one processor to access the memory space of another processor. Careful design and implementation of parallel programs in the explicit communication model can produce efficient parallel codes. However, programming

with explicit communication is proved to be tedious and error-prone, even using machine-independent interfaces such as PVM [3] and MPI [4]. This programming model is regarded as assembly language programming for parallel machines.

The distributed shared-memory model is based on the abstraction of virtual shared memory [5] built on physically distributed memories. Virtual shared memory can be implemented in hardware as in Stanford DASH machine or through software [6], providing an illusion of shared memory space. Programming with distributed shared-memory is easier than explicit communication. However, parallel programs in this model do not have enough information about data locality to facilitate compiler's optimization of memory access. The synchronization mechanism in this model usually includes barriers and locks. Barrier synchronization can be inflexible and is often stronger than necessary. Handling locks directly is tricky as shown in many thread packages such as Pthread [7] and Java [8].

In the data-parallel model [9], data domains are divided into sub-domains assigned to and operated on by different processors in parallel. The drawback of this model is that it offers little support for programming applications with irregular data sets. It is unlikely that it would become a mainstream model for general-purpose parallel programming in the future.

We believe that an ideal model for the mainstream general-purpose parallel and distributed programming should

- make programming easy (programmability),
- facilitate expressing maximum parallelism in applications (expression of parallelism), and
- support modular programming for large and complicated real-world applications (modularity).

Note that both shared-memory and explicit communication are abstractions close to machine architectures. It is hard to program at such a low level [2].

Interacting processes (IP) with multiparty interactions are a coordinated distributed programming model for interactive applications proposed by N. Francez and I. R. Forman [1]. The model has only three fundamental abstractions: process, synchronous multiparty interaction, and team and role. Processes are objects for concurrency. Synchronous multiparty interactions are objects to achieve *agreement* among concurrent processes. Agreement about values of variables and effect of synchronization is essential for reasoning about parallel and distributed programs[3]. Multiparty interactions allow processes to access non-local data with

[3] The authors of [1] believe that agreement is fundamental and more important than communication in distributed programming. They said:

> In order to appreciate the importance of multiparty interactions, one must understand the essence of the problem of designing and implementing concurrent systems. It is a mistake to think in terms of communication; instead, one must go beyond this narrow view and conceive the problem in terms of agreement. ... Of course, communication is part of the problem, but no language can rise to the challenge without recognizing that *agreement* is an abstraction

strong agreement. Combined with guard selection and iteration, they allow programmers to establish the agreement about the states of the parallel processes easily.

The support for modular distributed and parallel programming in IP is provided through *teams* and *roles*. A team is a module to encapsulate concurrency and interactions among the processes in it. It can be instantiated and referenced like an ordinary object. Roles in a team are formal processes to be enroled[4] by actual processes from outside of the team. Enrolements, like function calls in sequential programming models, can pass actual parameters. Therefore, teams can be used as parallel modules to build large parallel and distributed applications

The IP model is extremely powerful for programming distributed reactive applications. The purpose of this paper is to argue that IP is also an ideal model for parallel programming. In section 3, we present the IP programs for three typical problems in parallel computing to demonstrate the suitability of the model for parallel programming. We then provide the analysis and argument why the IP model is ideal for parallel programming in Section 4. We first introduce the IP model and its notations in Section 2. Section 5 concludes the paper with a short summary.

2 IP model and Notations

In this section, we introduce the IP programming model and its notations. More details of the model can be found in [1].

2.1 Teams and Processes

In IP, a program consists of a number of modules called *teams*, one of which is the main program. Each team consists of *processes* or *roles*. The main program contains only processes. A process in a team is a separate thread of execution and it starts to run as soon as the team is instantiated. A role is a formal process to be enroled by actual processes. It starts to run only when an actual process outside the team enroles it.

The processes (or roles) of the same type share the common code and are distinguished by the indices used as the process identifier. For example, we can use $\|_{i=0,n-1}$ **process** P_i to declare n processes of type P in a team. The code for processes of P can use the index i as the process identifier for the process. One way to implement the processes of the same type in a team is to create an array of threads and use the array index as the process identifier. The code of a type of process (role) follows the process (role) declaration. For example, the code for a team T which consists of n processes of type P and m roles of type R is written as follows:

---

that must be achieved.

We believe that this argument also applies to parallel programming.

[4] This is a new word suggested by Francez and Forman to mean "enter a role".

team $T()$ ::
[
  $\|_{i=0,n-1}$ **process** $P_i()$ ::
      ⋮ (the code for processes P)
  $\|_{j=0,m-1}$ **role** R_j ::
      ⋮ (the code for roles R)
]

The subscripted names such as P_i and R_j are introduced to simplify the presentation of IP pseudo codes. Of course, a team can declare many process (role) types, each of which can have multiple instantiations.

The code of each process or role type is a sequence of assignments, `for` and `if` statements[5], function and procedure calls from the sequential programming model as well as *interaction statements*, and enhanced CSP-like [10] guard selection and iteration statements.

2.2 Variables

A team can declare team variables which all processes (roles) declared in the team can read and write. Therefore, team variables are shared variables in the team. Team variables also include the formal parameters of the team to be bound to actual parameters when the team is instantiated.

Apart from team variables, each process or role can declare its own local variables. These variables are "local" in the sense that (1) they can be updated only by the process which declares them and (2) they reside in the local memory space of the process. Other processes or roles in the team can read them through interaction statements to be described shortly. This implies that the scope of these local variables is the entire team. These local variables are something between shared variables and strictly local variables with the scope being the declaring processes. You can call them "controlled shared variables" or "semi-local variables", but we simply call them local variables in IP.

Quite often the processes of the same type need to declare similar local variables. Again, we can use the process identifier to distinguish them. For example, n processes of type P can declare local variables with the same name k indexed by the process identifier i as follows:

  $\|_{i=0,n-1}$ **process** $P_i()$ ::
      int k_i;
      ⋮ (the code for processes P)

[5] The original IP notations do not have `for` and `if` statements. However, they can be easily realized by using guard iteration and selection statements in IP. We include them in the IP notations to improve readability.

Again, the subscripted names k_i are used to simplify the pseudo code presentation. They can be implemented by a distributed array across the processes and k_i is actually $k[i]$.

2.3 Interaction Statement

Within a team of an IP program, the synchronization and coordination among the concurrent processes or roles are captured by *multiparty interactions*. A multiparty interaction is defined by (1) a name and (2) p parties, each of which represents an *participant* of the interaction and is characterized by the code to be executed by the participant during the interaction. (Multiparty interactions are called interactions in the rest of the paper.) A process can *participate* in an interaction by executing an *interaction statement* of format $a[...]$, where a is the name of the interaction and the square brackets enclose the code to be executed by the participating process during the interaction.

An interaction a with p parties will not be executed until there are p processes ready to participate in it.

The code within the square brackets for each participating process of the interaction is a set of assignments whose left-hand sides are local variables of the process only. The right-hand expressions can contain non-local variables of other participating processes. The execution of the interaction is atomic, meaning that the right-hand expressions use the old values of the variables involved and the new values of the left-hand variables are visible only after the interaction is completed.

```
process P ::              process Q ::          process R ::
int x, y;                 int z;                int w;
   ⋮                         ⋮                     ⋮
x = 2;                    z = 3;                w = 4;
y = 1;                    a[z = w + y];         a[w = x + y - z];
a[x = z + w, y = x + z];  print z;              print w;
print x; print y;            ⋮                     ⋮
   ⋮
```

Fig. 1. Example of Multiparty Interaction

Figure 1 shows an example of three-party interaction named a in which processes P, Q and R participates. Processes P, Q and R have local variables $\{x, y\}$, $\{z\}$ and $\{w\}$, respectively. The three interaction statements of a are executed atomically in parallel when all the controls of P, Q and R reach them. The values of x and y printed by P after the interaction is 7 and 5, respectively. Note that the assignment to y uses the old value of x, 2, instead of the new value, 7.

Similarly, the values of z and w printed by Q and R after the interaction is 5 and 0, respectively.

The interaction in IP is an extension of rendezvous type of synchronization in Ada to allow synchronization among an arbitrary number of processes. It is also an extension of synchronous communication in Milner's CCS model [11] to allow synchronous communication among a group of arbitrary number of processes.

2.4 Guard Selection and Iteration

The IP model has CSP-like guard selection and iteration statements [10] enhanced with the interaction statements described above. The format of enhanced guard selection statements is as follows:

$$[B_1 \& a_1[...] \rightarrow S_1 \Box \cdots \Box B_n \& a_n[...] \rightarrow S_n]$$

In the k-th guard ($1 \leq k \leq n$), B_k is a boolean predicate of local variables called *guarding predicate* and $a_k[...]$ an interaction statement called *guarding interaction*; both are optional. S_k is a sequence of statements of any type.

A guard is *ready* if its guarding predicate is true. At least one guard must be ready in a guard selection statement. The guarding interaction of a ready guard is *enabled* if all the guards with the same interaction in other processes are also ready. An enabled guarding interaction can be selected for execution. If many guarding interactions in a guard selection statement are enabled, only one of them can be selected. Whether an enabled guarding interaction is actually selected depends on the result of coordination among the processes involved. If none of the enabled interactions is selected, the process is blocked. Figure 2

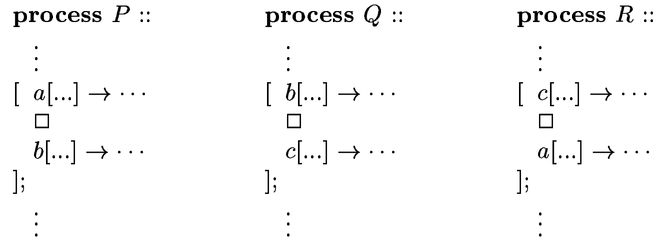


Fig. 2. Coordination for Selecting Enabled Interactions

shows an example of three guarding interactions, a, b and c, involving processes P, Q and R. Each of a, b and c has two parties. All the guarding interactions are enabled when P, Q and R are executing the guard selection statements, because there are no guarding predicates in the guards. However, only one of a, b and c can be selected for execution in this example. If a is selected, process Q is blocked and processes P and R execute it. Symmetrically, if b is selected to be executed by P and Q, R is blocked.

After a guarding interaction is selected and executed, the sequence of statements denoted S_k following the right arrow are executed.

The guard iteration statement is similar and of format:

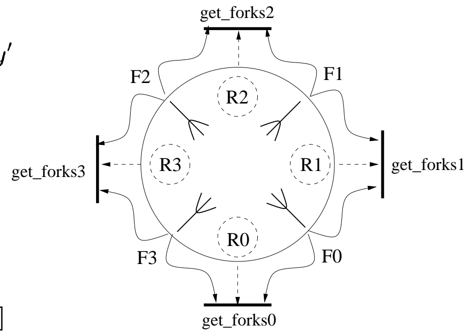
$$*[B_1\&a_1[...] \rightarrow S_1\Box \cdots \Box B_n\&a_n[...] \rightarrow S_n]$$

The difference is that the enclosed guard selection will be executed repeatedly until none of the guarding predicates is true.

As an example of guard iteration, a team to encapsulate the interactions in the dining philosophers problem is shown in Figure 3(a). In this problem, a dining philosopher sitting in a round dining table needs to pick up the two forks (or chopstics) at his/her both sides before he/she can eat. Each philosopher alternates between "thinking" and "eating". In this team, we use n processes,

team $TABLE(int\ n)$ ::
[
  $\|_{i=0,n-1}$ **role** $R_i()$ ::
    $char[\,]\ s_i =' thinking';$
    $*[\ (s_i ==' thinking') \rightarrow s_i =' hungry'$
      $\Box$
      $(s_i ==' hungry')\&$
        $get_forks_i[s_i =' eating']$
        $\rightarrow\ give_forks_i[\,]$
    ]
  $\|_{i=0,n-1}$ **process** F_i ::
    $*[\ get_forks_i[\,] \rightarrow\ give_forks_i[\,]$
      $\Box$
      $get_forks_{i+1}[\,] \rightarrow\ give_forks_{i+1}[\,]$
    ]
]

(a) Team of Dining Philosophers



(b) Three-party Interactions get_forks_i

Fig. 3. IP Module for Dining Philosophers Problem

F_i, to simulate the n forks, because they are part of the table and ready to be used as soon as the table is set up (instantiated). To simulate more general situations where possibly different philosophers can sit at the same table position at different times, we chose to use n roles, R_i, to code the behavior of philosopher and let philosopher processes outside the team enrole these roles.

There are n three-party interactions named get_forks_i and another n three-party interactions named $give_forks_i$ to encapsulate the interprocess synchronization[6].

[6] One weakness of IP syntax is that it lacks explicit declaration of multiparty interactions.

Again, we use subscripted names for the interactions here to simplify presentation. The participants of interaction get_forks_i and $give_forks_i$ are R_i, F_i and F_{i-1}[7]. Figure 3(b) illustrates the three-party interactions get_forks_i (thick bars) and their participating processes and roles.

When interaction get_forks_i is enabled and selected for execution, the philosopher process enroling (invoking) R_i gets both forks managed by F_i and F_{i-1} and eats. Since a philosopher process gets both forks in one atomic action, deadlock is not possible.

2.5 Roles and Their Enrolement

Roles in a team are formal processes to be enroled by actual processes. A role can be enroled by only one process at at a time. When multiple processes try to enrole a role, only one can succeed.

The code of the role is executed by the enroling process in the context of the team. The enroling process can pass parameters to the role, in much the same way as function calls with parameters. The enrolement is finished when the code of the role exits.

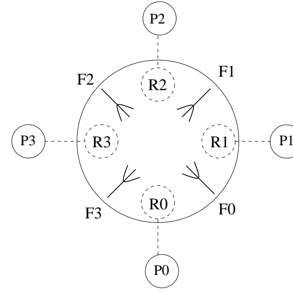
The format of enrolement statement is

$$< role_name > @ < team_designator > (< actual_parameters >)$$

Figure 4(a) shows an IP main program which creates a dining table team of size n and n philosopher processes to enrole the n roles in the team repeatedly. Figure 4(b) illustrates the enrolements with dash lines for the team of size 4.

team $MAIN$ (int n) ::
[
  team designator $table = TABLE(n)$;
  $\|_{k=0,n-1}$ **process** P_k ::
    `for (;;)` $R_k@table()$;
]

(a) Main Program



(d) Enrolements of Roles

Fig. 4. Simulation of Dining Philosophers Problem

We could create twice as many philosopher processes as the number of dining

[7] All the addition and subtraction in the indexes are modulo arithmetic operations with respect to n.

positions in the table and let two philosophers to compete to eat in the same position. The code would be as follows:

team $MAIN$ (int n) ::
[
  team designator $table = TABLE(n)$;
  $\|_{k=0,2n-1}$ **process** P_k ::
    `for (;;)` $R_{k \bmod n}@table()$;
]

3 IP Parallel Programming Examples

To demonstrate the suitability of IP for parallel programming, we present three IP parallel programs in this section. They are parallel sorting, parallel dynamic programming for optimal binary search tree and parallel successive over-relaxation (SOR). The algorithms for parallel sorting and parallel SOR are from [12]. The dynamic programming algorithm for optimal binary search tree is described in [13].

3.1 Parallel Sorting

To sort an array of n elements, $b[1..n]$, the odd-even transposition parallel sorting algorithm [12] goes through $\lceil n/2 \rceil$ iterations. In each iteration, each odd element $b[j]$ (j is odd) is compared with its even neighbor element $b[j+1]$ and the two are exchanged if they are out of order. Then each even element $b[i]$ (i is even) is compared and exchanged with its odd neighbor element $b[i+1]$ if necessary.

The algorithm can be easily implemented in IP with n parallel processes, each of which holds a data element. The compares and exchanges can be done through the interactions between neighbor processes.

The IP program for the parallel sorting is as follows, assuming the type of data to be sorted is `char`:

team $OESort(int\ n,\ char[\,]\ b)$ ::
[
  $\|_{j=0,n-1}$ **process** P_j ::
    $char\ a_j\ =\ b[j]$;
    `for (int i = 1; i <=` $\lceil n/2 \rceil$`; i++) {`
      //odd compare and exchange
      `if` $(j < n-1 \wedge odd(j))$
        $comp_j[a_j = \min(a_j, a_{j+1})]$;
      `else if` $(j > 0 \wedge even(j))$
        $comp_{j-1}[a_j = \max(a_{j-1}, a_j)]$;

      //even compare and exchange
      `if` $(j < n-1 \wedge even(j))$

```
            compj[aj = min(aj, aj+1)];
         else if (j > 0 ∧ odd(j))
            compj-1[aj = max(aj-1, aj)];
      }
      b[j] = aj;
]
```

Here, each process P_j holds a data element, a_j. The odd compares and exchanges are done through interactions $comp_j$ (j is odd, i.e. $odd(j)$ is true) between processes P_j and P_{j+1}. All these compares and exchanges can be done in parallel, because they use different interactions $comp_j$ (j is odd). The even compares and exchanges are similar.

Figure 5 illustrates the execution of this IP program for a `char` array: $b[\,] = \{'D','C','B','A'\}$. The boxes in the figure represent the three interactions, $comp_0$, $comp_1$ and $comp_2$, reused many times. The dotted vertical lines show the progress of the processes (from top to bottom).

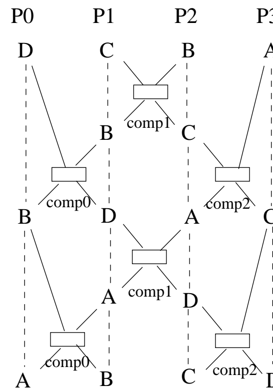


Fig. 5. Example of OESort

It can be proved that after passing through k pairs of odd and even interactions, the data held by each process is no farther than $n - 2k$ positions away from its final sorted position. After process P_j finishes $\lceil n/2 \rceil$ iterations, the data which it holds in a_j must have passed through $\lceil n/2 \rceil$ pairs of odd and even interactions. Therefore, the array is sorted when all processes are terminated.

Note that there is no global barrier synchronization across all processes in this program. The interactions and their synchronization are restricted between adjacent processes.

3.2 Optimal Binary Search Trees

As the second example, we present the IP program to find the optimal binary search tree (BST). Given n keys, $K_1 < \cdots < K_n$, with their probability of

occurrence, $p_1, \cdots, p_n$, the problem is to find the binary search tree for those keys so that the average search time is minimized.

The optimal binary search tree problem is best solved by dynamic programming [13]. The optimal BST with n keys can be obtained if the optimal BSTs with $n-1$ consecutive keys have been found. Let the optimal BST containing $j-i+1$ keys, $K_i, \cdots, K_j$ ($j \geq i-1$), be denoted $BST_{i,j}$ and its mean search time $MST_{i,j}$. Note that $BST_{i,i-1}$ is an empty tree and, thus, $MST_{i,i-1} = 0$. $BST_{1,n}$ can be obtained by comparing n binary search trees with the root nodes holding different keys K_l ($l = 1, \cdots, n$), and $BST_{1,l-1}$ and $BST_{l+1,n}$ as their left and right subtrees. The mean search time for $BST_{1,n}$ with root node K_l is

$$(MST_{1,l-1} + \sum_{k=1}^{l-1} p_k) + p_l + (MST_{l+1,l} + \sum_{k=l+1}^{n} p_k)$$

Therefore, the optimal mean search time $MST_{1,n}$ is as follows:

$$MST_{1,n} = \min_{1 \leq l \leq n} (MST_{1,l-1} + MST_{l+1,n}) + \sum_{k=1}^{n} p_k$$

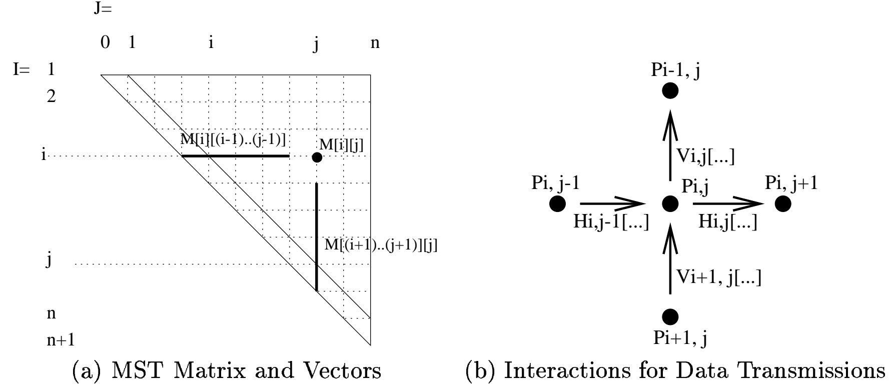
Subtrees $BST_{1,l-1}$ and $BST_{l+1,n}$ can be found recursively in the same way. In general, the formula to find $MST_{i,j}$ is as follows:

$$MST_{i,j} = \min_{i \leq l \leq j} (MST_{i,l-1} + MST_{l+1,j}) + \sum_{k=i}^{j} p_k \qquad (1)$$

The working data structure for finding $MST(1,n)$ is an $(n+1) \times (n+1)$ upper-triangular matrix M as shown in Figure 6. The matrix element $M[i][j]$ is used to store $MST(i,j)$ ($i \leq j$). Note that the index ranges of the first and second dimensions of M are $[1..(n+1)]$ and $[0..n]$, respectively. $M[k][k]$ ($1 \leq k \leq n$) and $M[k][k-1]$ ($1 \leq k \leq n+1$) are initialized to p_k and 0, respectively. The dynamic programming algorithm computes $M[i][j]$ ($1 \leq i < j \leq n$) by using vectors[8] $M[i][(i-1)..(j-1)]$ and $M[(i+1)..(j+1)][j]$ in M according to Equation (1). Figure 6(a) uses thick bars to show vectors $M[i][(i-1)..(j-1)]$ and $M[(i+1)..(j+1)][j]$, both of which contain $(j-i+1)$ elements. Obviously, the computations of $M[i][j]$ on the diagonal $j - i = k$ ($1 \leq k \leq n-1$) can be done in parallel, because they are data-independent. The result of the algorithm is stored in an $(n-1) \times (n-1)$ integer matrix R, where $R[i][j]$ ($2 \leq i \leq j \leq n$) is the index of the root node of optimal binary search tree $BST_{i,j}$.

We let each $MST_{i,j}$ be computed by a separate process $P_{i,j}$. The matrix M is implemented through local vectors in these processes. Each $P_{i,j}$ has two local vectors $h_{i,j}$ and $v_{i,j}$ both with $j-i+2$ elements. The index range of $h_{i,j}$ is $[(i-1)..j]$. Its sub-vector $h_{i,j}[(i-1)..(j-1)]$ is used to store $M[i][(i-1)..(j-1)]$

[8] Given a two-dimensional matrix $A[\,][\,]$, we use $A[i][j_1..j_2]$ to denote the sub-vector on the i-th row $(A[i][j_1], \cdots, A[i][j_2])$. Similarly, the sub-vector on the j-th column $(A[i_1][j], \cdots, A[i_2][j])$ is denoted $A[i_1..i_2][j]$.



(a) MST Matrix and Vectors (b) Interactions for Data Transmissions

Fig. 6. Data Structure and Interactions for Computing BST

and element $h_{i,j}[j]$ is used to store $M[i][j]$ computed. Similarly, the index range of $v_{i,j}$ is $[i..(j+1)]$. Its sub-vector $v_{i,j}[(i+1)..(j+1)]$ is used to store $M[(i+1)..(j+1)][j]$. Element $v_{i,j}[i]$ is the same as $h_{i,j}[j]$ and used to store $M[i][j]$.

Basically, each process $P_{i,j}$ does three things in sequence during its life time:

1. It inputs $M[i][(i-1)..(j-1)]$ from $P_{i,j-1}$ and $M[(i+1)..(j+1)][j]$ from $P_{i+1,j}$ and stores them in $h_{i,j}[(i-1)..(j-1)]$ and $v_{i,j}[(i+1)..(j+1)]$, respectively.
2. It computes $MST_{i,j}$ using (1) and stores it in both $h_{i,j}[j]$ and $v_{i,j}[i]$. It also stores the index of the root node of $BST_{i,j}$ in $R[i][j]$.
3. It outputs $h_{i,j}[(i-1)..j)]$ and $v_{i,j}[i..(j+1)]$ to $P_{i,j+1}$ and $P_{i-1,j}$, respectively.

The data transmissions from $P_{i,j}$ to $P_{i,j+1}$ and $P_{i-1,j}$ are done through the interactions named $H_{i,j}$ and $V_{i,j}$, respectively. Figure 6(b) illustrates the data transmissions in which $P_{i,j}$ is involved and the corresponding interactions used.

The IP program to compute the optimal binary search tree is as follows:

```
team OptimalBST(int n,
               float[1..n] p,
               float[1..(n+1)][0..n] M,
               int[2..n][2..n] R)
[
  ||i=1,n-1  j=2,n  j≥i+1 process P_{i,j} ::
    float[(i-1)..j] h_{i,j};
    float[i..(j+1)] v_{i,j};
    boolean h_in, v_in = false;
    boolean h_out, v_out = false;
    float mst; int root;

    if (j = i+1) { //take initial values
      h_{i,j}[(i-1)..(j-1)] = M[i][(i-1)..(j-1)];
      v_{i,j}[(i+1)..(j+1)] = M[(i+1)..(j+1)][j];
      }
```

```
else  //input from P_{i,j-1} and P_{i+1,j}
   *[ (j > i+1 ∧ ¬h_in) &
         H_{i,j-1}[h_{i,j}[(i-1)..(j-1)] = h_{i,j-1}[(i-1)..(j-1)] ]
            → h_in = true
      □
      (j > i+1 ∧ ¬v_in) &
         V_{i+1,j}[v_{i,j}[(i+1)..(j+1)] = v_{i+1,j}[(i+1)..(j+1)] ]
            → v_in = true
   ];

mst = min_{l=i}^{j}(h_{i,j}[l-1] + v_{i,j}[l+1]) + Σ_{k=i}^{j} p[k];
root = the value of l which makes the mst above;
h_{i,j}[j] = v_{i,j}[i] = mst;

//output local vectors to P_{i,j+1} and P_{i-1,j}
*[ (j < n ∧ ¬h_out) & H_{i,j}[ ] → h_out = true
   □
   (i > 1 ∧ ¬v_out) & V_{i,j}[ ] → v_out = true
];

R[i,j] = root; M[i,j] = mst;
]
```

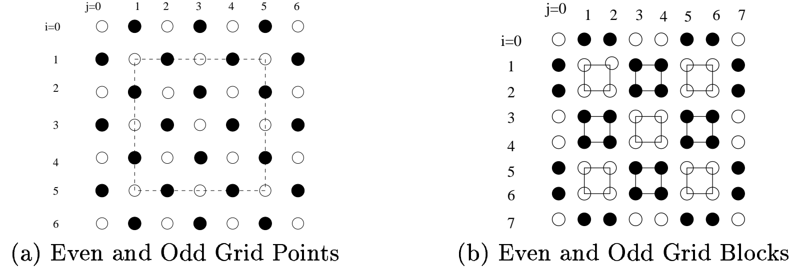
Note that the assignments in the square brackets of interactions $H_{i,j-1}$ and $V_{i+1,j}$ are vector assignment statements.

3.3 Parallel SOR

Successive Over-Relaxation (SOR) is a sequential iterative method to solve partial differential equations using finite difference. This method is also known as the Gauss Seidel method. It has faster convergence rate than the Jacobi method, which requires only $en/3$ iterations to reduce the error by a factor 10^{-e} for a problem of n^2 grid points. A parallel SOR algorithm called *odd-even ordering with Chebyshev acceleration* has the same convergence rate as the Gauss Seidel method [12].

Consider an $(n+2)\times(n+2)$ array $a[0..(n+1)][0..(n+1)]$ for the grid points in the parallel SOR. Each iteration has two phases to first update even grid points $a[i][j]$ ($i+j$ is even) and then odd grid points $a[i][j]$ ($i+j$ is odd), both using the values of neighbor grid points. Figure 7(a) uses hollow and filled circles to show the even and odd grid points, respectively. The updates in each phase can be done in parallel because there are no data dependencies between them. Since the computing formula and data access pattern are the same in both phases, it is better to encapsulate them in a common module. The IP model provides a perfect framework to do this through its parallel module, team. The IP program of the team to update even or odd grid points is as follows:

Team *Update*(*int n*)



(a) Even and Odd Grid Points (b) Even and Odd Grid Blocks

Fig. 7. Even and Odd Grid Points and Blocks

```
[
  ‖_{i=0,n+1  j=0,n+1} role W_{i,j}(float b, boolean eo) ::
    float a_{i,j} = b;
    boolean eastDone, southDone, westDone, northDone = false;

    if ( (eo ∧ even(i + j) ∨ ¬eo ∧ odd(i + j)) ∧ 0 < i, j < n + 1 )
      C_{i,j}[a_{i,j} = (a_{i-1,j} + a_{i,j-1} + a_{i+1,}a_{i,j+1}) / 4];        //update a_{i,j}
    else if (eo ∧ odd(i + j) ∨ ¬eo ∧ even(i + j) ) {

      //decide which neighbors to pass a_{i,j}
      if (i == 0) northDone = true;
      else if (i == n + 1) southDone = true;
      else if (j == 0) westDone = true;
      else if (j == n + 1) eastDone = true;

      // pass a_{i,j} to all neighbors
      *[(¬northDone ∧ i > 0) & C_{i-1,j}[ ] → northDone = true
      □
        (¬eastDone ∧ j < n + 1) & C_{i,j+1}[ ] → eastDone = true
      □
        (¬southDone ∧ i < n + 1) & C_{i+1,j}[ ] → southDone = true
      □
        (¬westDone ∧ j > 0) & C_{i,j-1}[ ] → westDone = true
      ];
    }
    b = a_{i,j};
]
```

There are $(n+2) \times (n+2)$ formal processes, $W_{i,j}$, each of which holds a grid point value $a_{i,j}$. Depending on the situation, $W_{i,j}$ either updates its $a_{i,j}$ using the grid points of its neighbors or contributes the value of $a_{i,j}$ to the neighbors for their grid points updates. The data exchange and computing for updating $a_{i,j}$ are done through interaction $C_{i,j}$. Boolean variables *eastDone, southDone, westDone* and *northDone* are used to make sure that the value of $a_{i,j}$ is used by all its neighbors before the code exits. If the argument *eo* of $W_{i,j}$ is true, even grid points are updated; otherwise odd grid points are updated.

The main IP program for the parallel SOR is as follows.

```
team ParallelSOR(int n, float[ ][ ] A, int e) ::
[
  //create a team of Update
  team designator update = new Update(n);

  ||i=0,n+1  j=0,n+1 process P_{i,j} ::
    float a_{i,j} = A[i, j];
    for (int k = 1; k <= ⌈en/3⌉; k++) {
      W_{i,j}@update(a_{i,j}, true);
      W_{i,j}@update(a_{i,j}, false)
    };
    if (0 < i, j < n + 1) A[i, j] = a_{i,j}
]
```

This program creates a team of *Update* and $(n+2) \times (n+2)$ processes $P_{i,j}$. During each iteration, process $P_{i,j}$ enroles $W_{i,j}$ of the team twice with same data argument $a_{i,j}$, but different boolean values for the second argument.

The IP program above can be easily extended to the blocked parallel SOR as illustrated in Figure 7(b). The grid points are grouped into blocks. The even and odd blocks are update alternatively. The updates of all even (odd) blocks can be done in parallel. The updates of grid points within each block are done sequentially as in the Gauss-Seidel method.

4 Suitability of IP for Parallel Programming

In this section, we analyze the IP programming model and show why it is a good model for parallel programming.

A good model for parallel programming should be easy to program, allow to express the maximum parallelism and support modular programming.

4.1 Ease-of-Programming

Since processes in IP are allowed to *read* non-local variables of other processes in a team, the union of the all local variables forms a shared name space. Since each variable can be updated only by its local process, data race is not possible in this shared name space. For instance, the local variables $a_{i,j}$ of all roles in team *Update* in the parallel SOR example form a shared name space for the grid points. As in the shared memory model [5], explicit interprocess data communication is not necessary.

Another feature of memory access in IP is that it restricts non-local variables accesses in interaction statements. All the code sections outside of interactions statements use only local variables and shared team variables. Combined with the synchrony and atomicity of the interaction statement in IP, this controlled

non-local variables access enables programmers to establish the agreement about values of all local variables easily. Consider the IP program for parallel SOR in Section 3.3 for instance. The programmer can easily establish the following agreement about all grid points values:

- Each grid point $a_{i,j}$ is updated $\lceil en/3 \rceil$ times, once for each iteration.
- The update of grid point $a_{i,j}$ by an even process in the k-th iteration uses the values of the grid points of its odd neighbors calculated in the $(k-1)$-th iteration (or the initial grid values if $k = 1$). The update of grid point $a_{i,j}$ by an odd process in the k-th iteration uses the values of the grid points of its even neighbors calculated in the same k-th iteration.
- The new value of each grid point $a_{i,j}$ calculated in an iteration is used to calculate the grid points of its all neighbors before it is updated in the next iteration.

Another reason for the ease-of-programming of IP is that it maintains the sequential programming model as a sub-model of computing for processes. With the agreement on the global view of the values of all local variables established by synchronous and atomic interaction statements, the reasoning about code sections outside interaction statements are purely sequential. We believe that this sub-model of sequential computation is very important for the ease of programming. While the real world is inherently parallel and distributed, each individual process in it is still sequential. Although some functions of human brains are parallel in nature (e.g., recognition of a human face), the reasoning process by human is mainly sequential. This is probably one of the reasons why some high-level parallel programming paradigms that conceal sequential computations (e.g. functional or logic programming) have never been widely accepted.

4.2 Parallelism and Efficiency

A good parallel programming model should also allow compilers to generate efficient parallel codes. To achieve this, the following qualities of the model are important.

- Programmers should be able to express the maximum parallelism in applications and algorithms.
- Programmers should be able to provide sufficient information about data distribution to enable compilers to optimize memory access and data communication.

Maximum Parallelism

The IP model supports the expression of maximum parallelism through its multiparty interaction and enhanced guard statements. Barrier synchronization [6, 14, 15] in the shared-memory parallel programming model is, in fact, a special case of multiparty interaction, where all the codes in the square brackets of the interaction statements are empty and the participants include all the parallel processes. Multiparty interaction is, therefore, more flexible and enforces

synchronization only among the relevant processes. In the both examples of parallel sorting and parallel SOR, synchronization occurs only among the processes that engage in the data flows and no global barrier synchronization is used. By using multiparty interactions we can minimize synchronization and maximize asynchrony in the parallel program. The cost of synchronization of multiparty interaction is also less than that of barrier synchronization involving all processes.

The IP model supports the expression of maximum parallelism also through its enhanced guard statement which allows enabled guarding interactions to be selected for execution in the nondeterministic order. For instance, in the parallel SOR example in Section 3.3 each process $P_{i,j}$ has to pass the value of its grid point $a_{i,j}$ to its neighbor processes through the interactions. The enhanced guard iteration statement does not specify any order of execution for these interactions. This allows the interactions to be completed in the shortest time. The same argument applies to the example of parallel optimal binary search tree in Section 3.2. Each process $P_{i,j}$ takes two inputs through interactions $H_{i,j-1}$ and $V_{i+1,j}$ before computing its $MST_{i,j}$. The enhanced guard iteration allows these inputs to take place in any order.

Memory Locality Information

Locality of memory access has a significant impact on the performance of parallel codes. In abstract high-level parallel programming models, the memory locality information is usually lost in the abstract address space such as virtual shared memory. The compilers have use to the run-time memory access patterns to optimize the memory access and it is a difficult task. As a result, abstract high-level models can hardly be implemented efficiently [2].

The IP model retains the memory locality information while providing the shared name space through restricted accesses in interaction statements. It seems to strike a good balance between the easy-of-programming through shared name space and the efficient compiler implementation.

4.3 Support for Modular Programming

Support for modular programming is essential for any programming model if it is to be used to develop large complicated applications. The IP model provides true parallel modules to encapsulate concurrency and interactions among the processes. Processes in a parallel module are first-class objects and can be parameterized. In the example of parallel SOR in Section 3.3 for instance, the parallel computation of new values for grid points as well as the interaction and data communication among the neighbor processes are all encapsulated in a parallel module which is used twice in each iteration.

The parallel modules in the IP model allow parallel algorithms and design patterns to be re-used. This enables parallel programmers to develop large complicated parallel applications in accordance with the well-established software engineering methodologies.

5 Conclusion

We have presented three parallel programs in the IP programming model to demonstrate the suitability of this model for parallel programming. We analyze the IP model and provide the new insights into it from the parallel programming perspective. We have shown that that IP with multiparty interaction is an ideal model for parallel programming. It suitability lies in its ease-of-programming, its capability to express the maximum parallelism and its support for modular parallel programming.

References

1. Nissim Francez and Ira R Forman. *Interacting Processes – A Multiparty Approach to Coordinated Distributed Programming.* Addison-Wesley, 1996.
2. D.B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computer Surveys*, 30(2):123–169, June 1998.
3. A. Geist, A. Beguelin, J. Dongarra, and W. Jiang at el. *PVM: Parallel Virtual Machine - A User guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.
4. W. Gropp, E. Lusk, and A. Skjellum. *MPI: Portable Parallel Programming with the Massage Passing Interface.* MIT Press, 1994.
5. K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, 1989.
6. C. Amza, A.L. Cox, S. Dwarkadas, and P. Keleher at el. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.
7. Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. *Pthreads Programming: POSIX Standard for Better Multiprocessing.* O'Reilly Associates, Inc., 1996.
8. Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns.* Addison-Wesley Longman, Inc., 1996.
9. High Performance Fortran Forum. High performance fortran language specification. *Scientific Programming*, 1(1-2):1–170, 1993.
10. C.A.R. Hoare. *Communication Sequential Processes.* Prentice Hall, 1985.
11. Robin Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
12. Michael J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill Book Company, 1987.
13. C.H. Nevison, D.C. Hyde, G.M. Schneider, and P.T. Tymann. *Laboratories for Parallel Computing.* Jones and Bartlett Publishers, 1994.
14. Peter Carlin, Mani Chandy, and Carl Kesselman. The compositional c++ language definition. Technical Report http://globus.isi.edu/ccpp/lang_def/cc++-def.html, California Technology Institute, 1993.
15. Parallel Fortran Forum. Parallel fortran from x3h5, version 1. Technical report, X3H5, 1991.

This article was processed using the LaTeX macro package with LLNCS style