Performance Results of Running Parallel Applications on the InteGrade[*]

Edson Norberto Cáceres, Henrique Mongelli,
Leonardo Loureiro, Christiane Nishibe
Dept. de Computação e Estatística
Universidade Federal de Mato Grosso do Sul
Campo Grande - MS, Brazil

Siang Wun Song
Dept. de Ciência da Computação
Universidade de São Paulo
São Paulo - SP, Brazil

Abstract

The InteGrade project is an on-going project with the participation of several research groups in Brazil. It is an opportunistic grid middleware that intends to exploit the idle time of computing resources in computer laboratories. The proposed middleware is already operational and the present work has the objective of studying the performance of parallel applications with communication among processors. We present two algorithms to evaluate the performance of the InteGrade middleware. The applications running under the InteGrade grid take slightly more time than those running under the standard MPI on a cluster. The results are satisfactory. The overhead of the InteGrade middleware is acceptable, in view of the benefits obtained to ease the use of grid computing by the user.

1. Introduction

The InteGrade Project [11, 13] aims the construction of a middleware that allows the implementation of a computing grid with non-dedicated computing resources, by using the idle capacity usually available in already installed computer laboratories. The InteGrade is a project developed jointly by researchers of several institutions: Department of Computer Science of Universidade de São Paulo, Departments of Informatics of Pontifícia Universidade Católica (Rio de Janeiro), Universidade Federal do Maranhão and Department of Computing and Statistics of Universidade Federal de Mato Grosso do Sul.

InteGrade has an object oriented architecture, where each module of the system communicates with the other modules through remote method invocations. InteGrade uses CORBA [12] as its infrastructure of distributed objects, thus benefiting from an elegant and solid architecture. This results in the ease of implementation, since the communication with the system modules is abstracted from the remote method invocations.

InteGrade was designed with the objective of allowing the development of applications to solve a broad range of problems in parallel. Several grid computing systems restrict their use to problems that can be decomposed into independent tasks, such as *Bag-of-Tasks* [6] or parametric applications. In addition to handling bag-of-tasks applications, InteGrade also aims to deal with parallel applications with dependencies that require communication among processors. To this end we designed parallel algorithms for several applications such as the 0-1 Knapsack Problem and the string alignment problem where communication is required.

One question that arises when one uses grid computing is the overhead of the grid middleware that ensures an integrated environment with special modules to handle the job submission, checkpointing, security, task migration, etc., in contrast to running a parallel algorithm in a cluster without such a middleware to ease the concern of the user. We compare the execution on a cluster with only MPI support and on a grid using the InteGrade middleware and MPI. Our results show a slight performance degradation when the parallel applications are run on the InteGrade. The difference in performance with respect to running on a cluster is, however, small. This is encouraging and shows a small and acceptable overhead of the InteGrade middleware.

2 Coarse-Grained Multicomputer Model

We use a version of the BSP model [21] referred to as the *Coarse-Grained Multicomputer* (CGM) model [7]. Due to the similarity we also use the term BSP/CGM. It uses only two parameters: the input size n and the number of processors p. Let N denote the input size of the problem. A BSP/CGM consists of a set of p processors each with local memory and each processor is connected by a router that can send messages in a point-to-point fashion.

[*]Partially supported by FAPESP Proc. No. 2004/08928-3 CNPq Proc. Nos. 55.0895/07-8, 30.5362/06-2, 30.2942/04-1, 62.0123/04-4, 48.5460/06-8, 62.0171/06-5 and FUNDECT 41/100.115/2006.

A BSP/CGM algorithm consists of alternating local computation and global communication rounds separated by a synchronization barrier.

In a computing round, we usually use the best sequential algorithm in each processor to process locally its data. In each communication round the total data exchanged by each processor (sends/receives) is limited by $O(N/p)$. We require that all information sent from a given processor to another processor in one communication round be packed into one long message, thereby minimizing the message overhead. In the BSP/CGM model, the communication cost is modeled by the number of communication rounds. The goal is to minimize the number of communication rounds as well as the total local computation time.

3. The 0-1 Knapsack Problem

The 0-1 Knapsack Problem can be formulated as follows. Let $S = \{1, 2, \ldots, n\}$ be a set of n distinct items such that the ith item is worth v_i dollars and weighs w_i kilos, where v_i and w_i are integers. Let W be the knapsack of integer capacity W used to carry the items. The question is: which items should be selected in order to fill the knapsack with the most valuable load without exceeding the capacity constraint, i.e.

$$\max\{\sum_{i=1}^{n} v_i z_i : \sum_{i=1}^{n} w_i z_i \le W, z_i \in \{0,1\}\}.$$

This problem belongs to the class of NP-*complete* problems [9]. However it is known that this problem can be solved sequentially in $O(nW)$ time. This time bound in not polynomial in the size of the input since $\lg W$ bits are required to encode the input W. We call this solution *pseudo-polynomial* [9]. There are two basic approaches for finding the exact solutions of the 0-1 Knapsack Problem: *dynamic programming* (DP) and *branch-and-bound* (B&B). When the parameters v_i and w_i are independently generated and we have large-size problems, the B&B approach is more efficient on the average for serial machine implementations [14]. When these parameters are correlated the DP approach behaves better than B&B [4, 5]. We will present a BSP/CGM algorithm that is based on DP approach.

The first Knapsack algorithm based on dynamic programming approach was developed by Gilmore and Gomory [10].

Consider the 0-1 Knapsack Problem with set of objects $[1, r]$ and weight c. Denote the value of the optimal solution for this problem by $f(r, c)$, with $1 \le r \le n$ and $0 \le c \le W$. Thus, $f(n, W)$ is the value of the optimal solution. The recurrence relation is:

$$f(r, c) = \max\{f(r-1, c), f(r, c-w_r) + v_r\}$$

$\forall c$, with $0 \le c \le W$, where $r = 1, 2, \ldots n$.

Algorithm 1 solves sequentially the 0-1 Knapsack Problem in $O(nW)$ time.

Algorithm 1 SEQUENTIAL 0-1 KNAPSACK ALGORITHM

```
Input: (1) $v_i$ and $w_i$, $1 \le i \le n$; (2) $W$; and (3) $p$.
Output: $f(n, W)$
for $c \leftarrow 1$ to $W$ do
    $f(0, c) \leftarrow 0$;
end for
for $r \leftarrow 1$ to $n$ do
    for $c \leftarrow 1$ to $W$ do
        if $c < w_k$ then
            $f(r, c) \leftarrow f(r-1, c)$;
        else
            $f(r, c) \leftarrow \max\{f(r, c-w_r) + v_k, f(r-1, c)\}$;
        end if
    end for
end for
```

Parallel algorithms on several parallel computing models for this problem have been proposed by [2, 4, 8, 15, 20].

3.1. The Wavefront Algorithm

We present a BSP/CGM algorithm for the 0-1 Knapsack Problem that is based on the wavefront paradigm of [1]. A characteristic and advantage of the wavefront or systolic paradigm is the modest communication requirement in the sense that each processor communicates with few other processors. This makes it very suitable as a potential application for grid computing.

In this section we present an $O(p)$ communication rounds BSP/CGM algorithm for computing the solution of the 0-1 Knapsack Problem with n items and maximum weight W. We will use p processors, where each processor has $O(Wn/p)$ local memory.

Let us first give the main idea to compute the optimal solution matrix f by p processors. For the set $S = \{1, 2, \ldots, n\}$ of items, the array w, where $w[i]$ is the weight of item i, is broadcasted to all processors, and the array v, where $v[i]$ is the value of item i, divided into p pieces, of size n/p, and each processor P_i, $1 \le i \le p$, receives the i-th piece of v ($v[(i-1)n/p + 1 \,..\, in/p]$).

The scheduling scheme can be illustrated in Figure 1. The notation P_i^k denotes the work of processor P_i at round k. Thus initially P_1 starts computing at round 0. Then P_1 and P_2 can work at round 1, P_1, P_2 and P_3 at round 2, and so on. In other words, after computing the k-th part of the sub-matrix f_i (denoted f_i^k), processor P_i sends to processor P_{i+1} the elements of the right boundary (rightmost column) of f_i^k. These elements are denoted by R_i^k. Using R_i^k, processor P_{i+1} can compute the k-th part of the sub-matrix f_{i+1}. After $p-1$ rounds, processor P_p receives R_{p-1}^1 and computes the first part of the sub-matrix f_p. In the $2p-2$ round, processor P_p receives R^p_{p-1} and computes the p-th part of the sub-matrix f_p and finishes the computation.

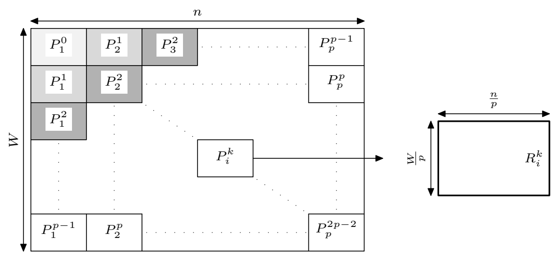

Figure 1. An $O(p)$ communication rounds scheduling

It is easy to see that with this scheduling, processor P_p only initiates its work when processor P_1 is finishing its computation, at round $p-1$. The load is thus unbalanced. A complet analysis about parameters to balance load can be found in [3].

The parallel algorithm is shown in Algorithm 2.

Algorithm 2 PARALLEL 0-1 KNAPSACK ALGORITHM

Input: (1) The number p of processors; (2) The number i of the processor, where $1 \leq i \leq p$; and (3) The array w, the capacity of the knapsack W and subarray v_i of size $\frac{n}{p}$, respectively.

Output: $f(r,c) = \max\{f[r, c-w[r]]+v[r], f[r-1,c]\}$, where $1 \leq c \leq W$ and $(j-1)\frac{n}{p}+1 \leq r \leq j\frac{n}{p}$.

```
1:  for $1 \leq k \leq p$ do
2:    if $i = 1$ then
3:      for $(k-1)\frac{W}{p}+1 \leq r \leq k\frac{W}{p}$ and $1 \leq c \leq \frac{n}{p}$ do
4:        compute $f(r,c)$;
5:      end for
6:      send($R^k_i$,$P_{i+1}$);
7:    end if
8:    if $i \neq 1$ then
9:      receive($R^k_{i-1}$, $P_{i-1}$);
10:     for $(k-1)\frac{W}{p}+1 \leq r \leq k\frac{W}{p}$ and $1 \leq c \leq \frac{n}{p}$ do
11:       compute $f(r,c)$;
12:     end for
13:     if $i \neq p$ then
14:       send($R^k_i$,$P_{i+1}$);
15:     end if
16:   end if
17: end for
```

3.2. Experimental Results

We have run the BSP/CGM 0-1 knapsack algorithm on a cluster composed by 12 nodes. This cluster is consisted of 6 CPU Intel Pentium IV of 1.7GHz and 6 CPU AMD Athlon of 1.6GHz, besides the nodes are connected by a 1Gb fast-Ethernet switch. The data used in the tests were generated randomly.

The 0-1 Knapsack parallel algorithm is implemented using standard ANSI C. On the cluster we used LAM-MPI library while on the cluster used as an InteGrade grid we used the InteGrade middleware and MPI. The purpose of the experiment is to compare the two executions, on the cluster using LAM-MPI and on the grid using the InteGrade middleware and MPI.

Table 1 and Figure 2 show the running times (in seconds) for the 0-1 Knapsack parallel algorithm running on the cluster using LAM-MPI.

$W \times n$	p=1	p=2	p=4	p=8
4096×1024	0.071	0.063	0.057	0.050
8192×2048	0.283	0.250	0.244	0.173
16384×4096	1.105	0.992	0.952	0.645
32768×8192	4.050	3.953	3.718	2.390

Table 1. Running times for 0-1 Knapsack on the cluster using LAM-MPI

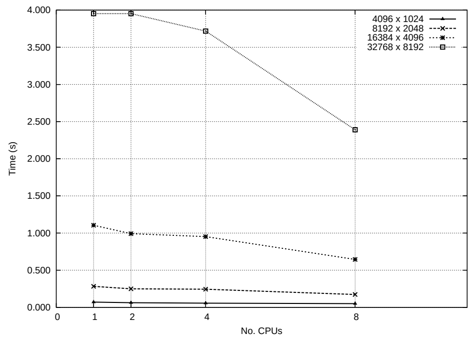

Figure 2. Running times for 0-1 Knapsack on the cluster using LAM-MPI

Table 2 and Figure 3 show the running times (in seconds) for the 0-1 Knapsack parallel algorithm running on the grid using InteGrade middleware and MPI.

$W \times n$	p=1	p=2	p=4	p=8
4096×1024	0.084	0.072	0.078	-
8192×2048	0.367	0.278	0.280	-
16384×4096	1.105	1.053	1.146	-
32768×8192	4.591	4.065	4.079	-

Table 2. Running times for 0-1 Knapsack on the grid using InteGrade MPI

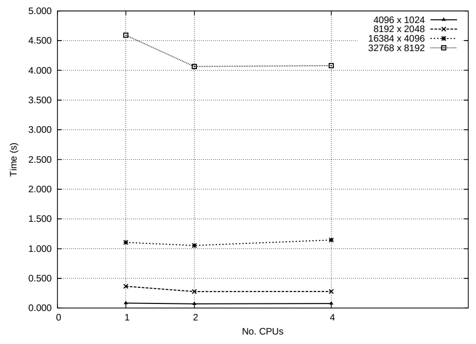

Figure 3. Running times for 0-1 Knapsack on the grid using InteGrade MPI

	4096×1024		8192×2048	
p	I	II	I	II
1	0.071	0.084	0.283	0.367
2	0.063	0.072	0.250	0.278
4	0.057	0.078	0.244	0.280
8	0.050	-	0.173	-

Table 3. Comparing running times for the 0-1 Knapsack Problem

	16384×4096		32768×8192	
p	I	II	I	II
1	1.105	1.105	4.050	4.591
2	0.992	1.053	3.953	4.065
4	0.952	1.146	3.718	4.079
8	0.645	-	2.390	-

Table 4. Comparing running times for the 0-1 Knapsack Problem

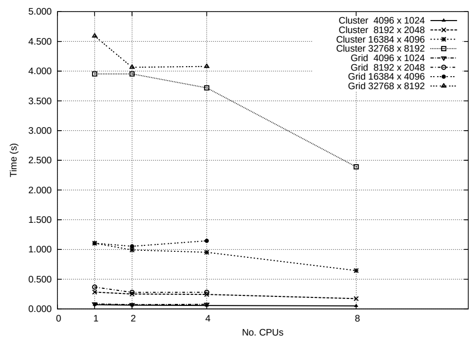

Figure 4. Compare running times for 0-1 Knapsack on the cluster and on the grid

Tables 3 and 4 and Figure 4 present a comparison between the running times on a cluster using standard LAM-MPI and on the grid running the InteGrade middleware and MPI. Column I and column II show the times on the cluster and on the grid, respectively. Figure 4 shows the corresponding curve.

We observe that the running time on the cluster using only LAM-MPI without the InteGrade middleware is slightly better than the times on the grid. Only in one case the times are the same. Notice that in the grid, the InteGrade middleware determines the choice of the machines.

4. Local Alignment Problem

The Local Alignment Problem is defined as follows. Given two sequences S_1 and S_2 over a given alphabet, find a subsequence of S_1 similar to a subsequence of S_2 under a given similarity metric. In Biology, the local alignment is used to determine if two sequences of nucleotides or proteins have similar functionality or evolutionary relationship.

Algorithm 3 SEQUENTIAL LOCAL ALIGNMENT ALGORITHM

```
Input: (1) Sequences $S_1$ and $S_2$, (2) $h$: penalty to start a gap, (3) $g$: penalty to extend a gap
Output: Best local alignment between $S_1$ and $S_2$
1: $A(S_1.length + 1, S_2.length + 1)$; // matrix to align one element of $S_1$ with one element of $S_2$
2: $B(S_1.length + 1, S_2.length + 1)$; // matrix to align one element of $S_1$ with a gap
3: $C(S_1.length + 1, S_2.length + 1)$; // matrix to align one element of $S_2$ with a gap
4: for $i \leftarrow 0$ to $S_1.length$ do
5:     $A[i, 0] \leftarrow 0$;
6:     $B[i, 0] \leftarrow 0$;
7:     $C[i, 0] \leftarrow 0$;
8: end for
9: for $j \leftarrow 0$ to $S_2.length$ do
10:    $A[0, j] \leftarrow 0$;
11:    $B[0, j] \leftarrow 0$;
12:    $C[0, j] \leftarrow 0$;
13: end for
14: for $i \leftarrow 0$ to $S_1.length$ do
15:    for $j \leftarrow 0$ to $S_2.length$ do
16:        $A[i, j] \leftarrow max(A[i-1, j-1], B[i-1, j-1], C[i-1, j-1]) + BLOSUM62[S1[i-1], S2[j-1]]$;
17:        $B[i, j] \leftarrow max(-(h+g) + A[i, j-1], -g + B[i, j-1], -(h+g) + C[i, j-1])$;
18:        $C[i, j] \leftarrow max(-(h+g) + A[i, j-1], -(h+g) + B[i, j-1], -g + C[i, j-1])$;
19:    end for
20: end for
```

The local alignment is a special case of the sequence alignment problem, which aligns two or more sequences through the insertion of gaps (holes) in order to achieve the highest degree of similarity between the sequences. There are two forms of sequence alignment: global and local. In the global alignment we want to achieve the highest degree of similarity of the entire sequence; in the local alignment we consider the similarity between subsequences.

To solve the Local Alignment Problem we use the Smith-Waterman algorithm [19], which is a variation of the global alignment algorithm of Needleman-Wunsch [17].

For two sequences of size m and n the algorithms runs in $O(m \times n)$ time with $O(m \times n)$ space. To align two sequences, the algorithm considers three possibilities [16]:

1. Align one element of S_1 with one element of S_2;
2. Align one element of S_1 with a gap;
3. Align one element of S_2 with a gap.

For each case the algorithm uses an $m \times n$ matrix and saves the case which gives a better alignment. We refer the reader to the book by Setubal and Meidanis [18] for details of the sequential algorithm. A substitution matrix is used to score an alignment between two elements. In our tests was used the BLOSUM62 substitution matrix, and for scoring the alignment of an element with a gap was used a linear function.

Algorithm 4 PARALLEL LOCAL ALIGNMENT ALGORITHM

```
Input: (1) Sequences $S_1$ of size $m$ and $S_2$ of size $n$, (2) Number
    of processors $p$ (3) Rank of processor $i$ (3) Each processor of
    rank $i$ holds $s1[0..m-1]$ and $s2[i*(n/p)..(i+1)*(n/p)]$
Output: Best local alignment between $S_1$ and $S_2$
    matrix A(m+1, blockSize+1), matrix B(m+1, blockSize+1),
    matrix C(m+1, blockSize+1)
 1: $blockSize \leftarrow n/p$
 2: $next \leftarrow i+1$
 3: $previous \leftarrow i-1$
 4: $col \leftarrow 1$
 5: for $round \leftarrow 0$ to $p-1$ do
 6:     $col \leftarrow col + blockSize$
 7:     if $i \neq 0$ then
 8:         receive $(A[0, col..col+blockSize], previous)$
 9:         receive $(B[0, col..col+blockSize], previous)$
10:         receive $(C[0, col..col+blockSize], previous)$
11:     end if
12:     compute $A[1..m, col..col+blockSize]$
13:     compute $B[1..m, col..col+blockSize]$
14:     compute $C[1..m, col..col+blockSize]$
15:     if $i \neq p-1$ then
16:         send $(A[m, col..col+blockSize], next)$
17:         send $(B[m, col..col+blockSize], next)$
18:         send $(C[m, col..col+blockSize], next)$
19:     end if
20: end for
```

4.1. The Parallel Algorithm

We present an $O(p)$ communication round and $O(m \times n/p)$ time BSP/CGM algorithm using p processors to compute the local alignment of two sequences S_1 and S_2 of sizes m and n, respectively. The parallel algorithm is shown in Algorithm 4.

4.2. Experimental Results

We ran the parallel local alignment algorithm on the cluster using LAM-MPI and on the grid running InteGrade middleware and MPI. Again the purpose of the experiment is to compare the two executions, on the cluster using LAM-MPI and on the grid using the InteGrade middleware and MPI.

Figure 5 shows the running times (in seconds) on the cluster using LAM-MPI.

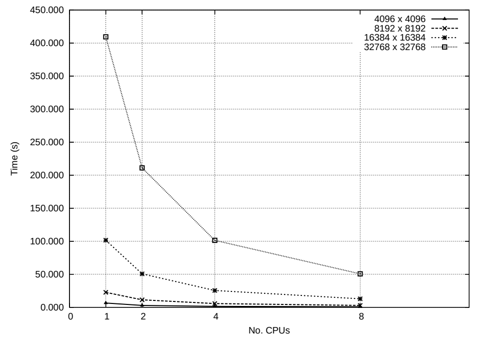

Figure 5. Running Times for Local Alignment on the cluster using LAM-MPI

Figure 6 shows the running times (in seconds) on the grid using InteGrade MPI.

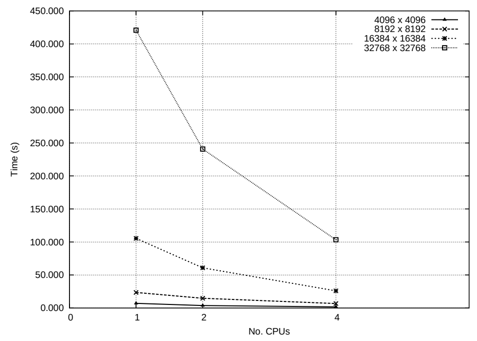

Figure 6. Running times for Local Alignment on the grid InteGrade MPI

Figure 7 shows the corresponding curve comparing compares the running times on the cluster using standard LAM-

MPI and on the grid running the InteGrade middleware and MPI.

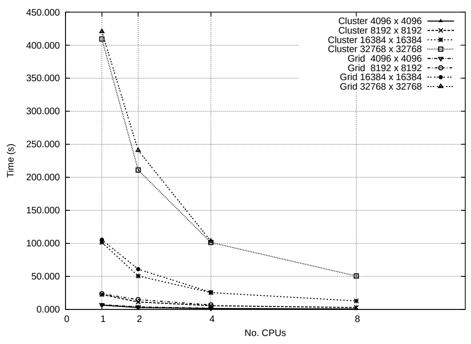

Figure 7. Comparing running times for Local Alignment on the cluster and on the grid

5. Conclusions

The present work has the objective of studying the performance of running parallel applications with communication among processors under the InteGrade middleware. We presented two algorithms to evaluate the performance of the InteGrade middleware. The applications running under the InteGrade grid take slightly more time than those running under the standard MPI in a cluster. The results are considered to be satisfactory, since the time difference is not substantial. This shows the overhead of the InteGrade middleware is acceptable, in view of the benefits obtained to ease the use of grid computing by the user.

References

[1] C. E. R. Alves, E. N. Cáceres, F. Dehne, and S. W. Song. A Parallel Wavefront Algorithm for Efficient Biological Sequence Comparison. In *Proceedings ICCSA 2003*, volume 2668 of *Lecture Notes in Computer Science*, pages 249–258. Springer, 2003.

[2] R. Andonov, F. Raimbault, and P. Quinton. Dynamic Programming Parallel Implementations for Knapsack Problem. Technical Report RI 740, IRISA, 1993.

[3] E. N. Cáceres and C. Nishibe. 0-1 Knapsack Problem: BSP/CGM Algorithm and Implementation. In *IASTED PDCS*, pages 331–335, 2005.

[4] G. Chen, M. Chern, and J. Jang. Pipeline Architectures for Dynamic Programming Algorithms. *Parallel Computing*, 13:111–117, 1990.

[5] C. Chung, , M. S. Hung, and W. O. Rom. A Hard Knapsack Problem. *Naval Research Logistics*, 35:85–98, 1988.

[6] D. P. da Silva, W. Cirne, and F. V. Brasileiro. Trading Cycles for Information: Using Replication to Schedule Bag-of-Tasks Applications on Computational Grids. *Lecture Notes in Computer Science*, 2790:169–180, 2003.

[7] F. Dehne. Coarse Grained Parallel Algorithms. *Special Issue of Algorithmica*, 24(3/4):173–176, 1999. Editorial Note.

[8] A. Ferreira and J. M. Robson. Fast and Scalable Parallel Algorithms for Knapsack-like Problems. *J. Parallel Distrib. Comput.*, 39:1–13, 1996.

[9] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[10] P. C. Gilmore and R. E. Gomory. The Theory and Computation of Knapsack Functions. *Operations Research*, 14:1045–1074, 1966.

[11] A. Goldchleger, F. Kon, A. Goldman, M. Finger, and G. C. Bezerra. InteGrade: object-oriented Grid middleware leveraging the idle computing power of desktop machines. *Concurrency and Computation: Practice and Experience*, 16(5):449–459, March 2004.

[12] O. M. Group. *CORBA v3.0 Specification*. Needham, MA, July 2002. OMG Document 02-06-33.

[13] InteGrade. http://gsd.ime.usp.br/integrade, 2004.

[14] S. Martello and P. Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[15] D. Morales, J. Roda, F. Almeida, C. Rodrigues, and F. Garcia. Integral Knapsack Problems: Parallel Algorithms and theirs Implementations on Distributed Systems. In *Proc. of the ACM-ICS 95*, pages 218–226, 1995.

[16] T. Rognes and E. Eeberg. Six-fold Speed-up of Smith-Waterman Sequence Database Searches Using Parallel Processing on Common Microprocessors. *Bioinformatics*, 16(8):699–706, 2000.

[17] C. D. W. Saul B. Needleman. A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970.

[18] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.

[19] T. F. Smith and M. S. Waterman. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147(1):195–197, March 1981.

[20] S. Teng. Adaptative Parallel Algorithm for Integral Knapsack Problems. *J.of Parallel and Distributed Computing*, 14:1045–1074, 1990.

[21] L. Valiant. A Bridging Model for Parallel Computation. *Communication of the ACM*, 33(8):103–111, 1990.