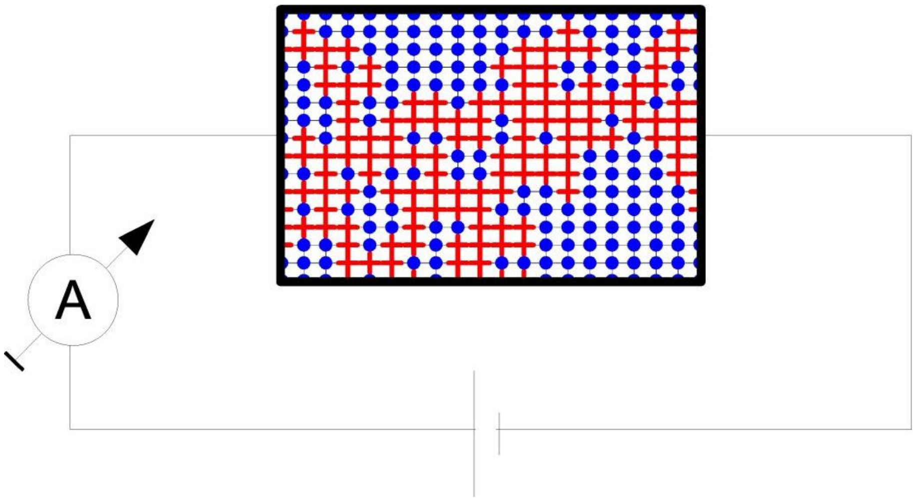
Percolation Study of samples on 2D lattices using GPUs

D.A Matoz-Fernandez, P.M Pasinetti and A.J Ramirez Pastor

Universidad Nacional de San Luis
Instituto de Física Aplicada
CONICET

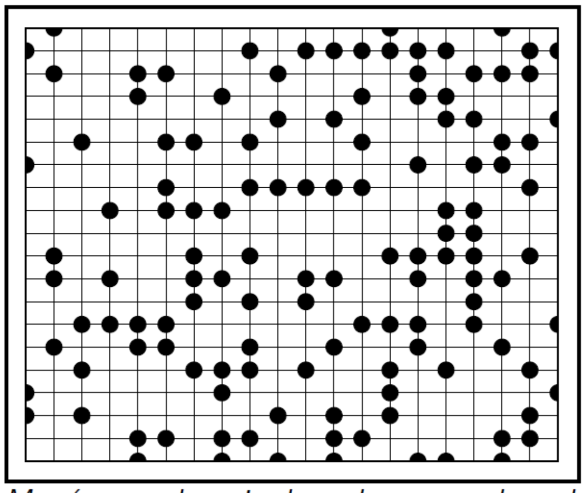
The percolation Problem



Application Fields

- Fluids
- Polymers Chemistry and Physics
- Rains drops
- Confinement of quarks in atomic nuclei
- Stars formations
- Image processing
- Fracture analysis

The percolation Problem

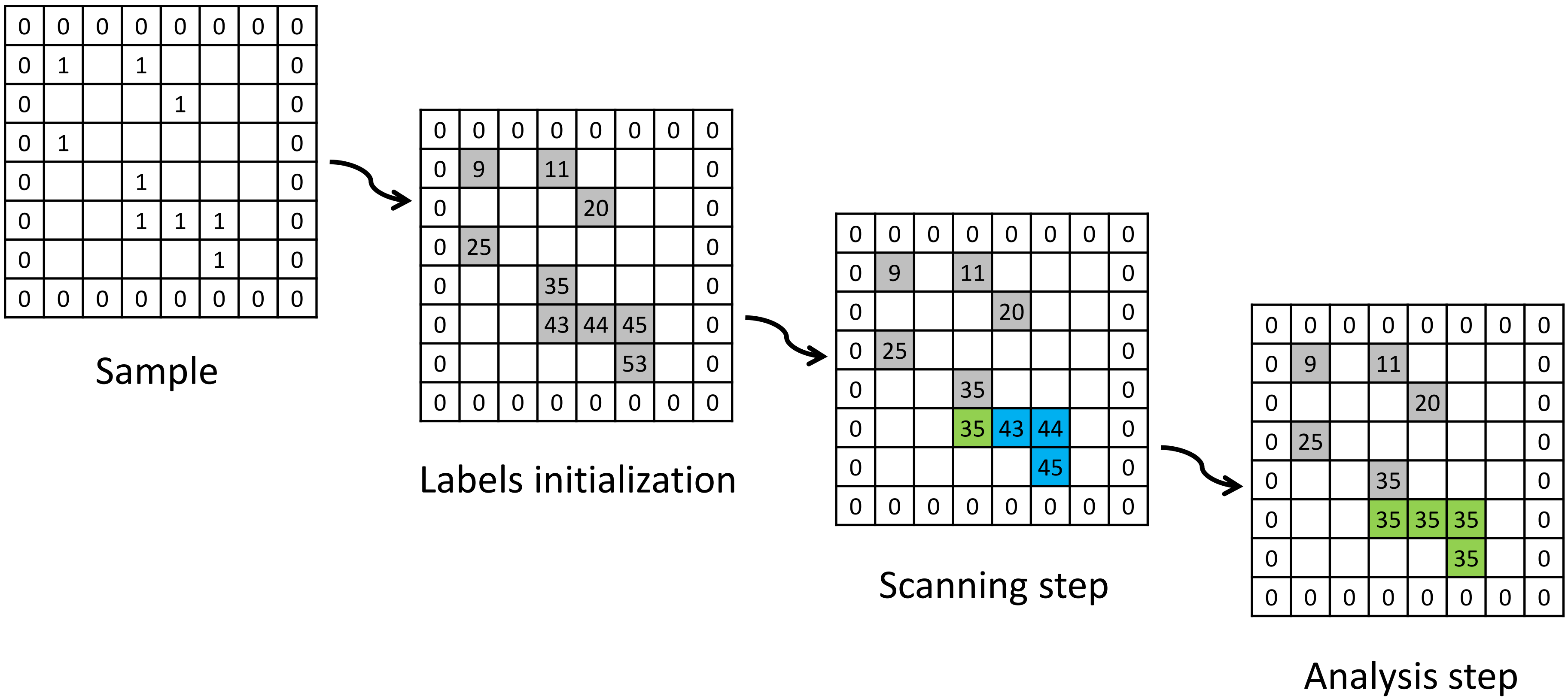


Application Fields

- Fluids
- Polymers Chemistry and Physics
- Rains drops
- Confinement of quarks in atomic nuclei
- Stars formations
- Image processing
- Fracture analysis

is a purely geometric phenomena!

The Algorithm



Label equivalence procedure

From left to right: original sample, initial labels assignation, labels after a Scanning step, and finally the situation after an Analysis step.

K.A. Hawick, A. Leist and D.P. Playne, Parallel Computing 36(12),655-678(2010).
Oleksandr Kalentev, Abha Rai, Stefan Kemnitz, Ralf Schneider, J. Parallel Distrib. Comput. 71(4): 615-620 (2011).

Simulation in Physics

Percolation Problem

```
FOR i=1 to Number of Samples
{
    0-Generate the sample;
    1-Percolation detection algorithm;
    2-Calculate  the physics variables of interest;
    3-Calculate Averages;
}
END
```

Simulation in Physics

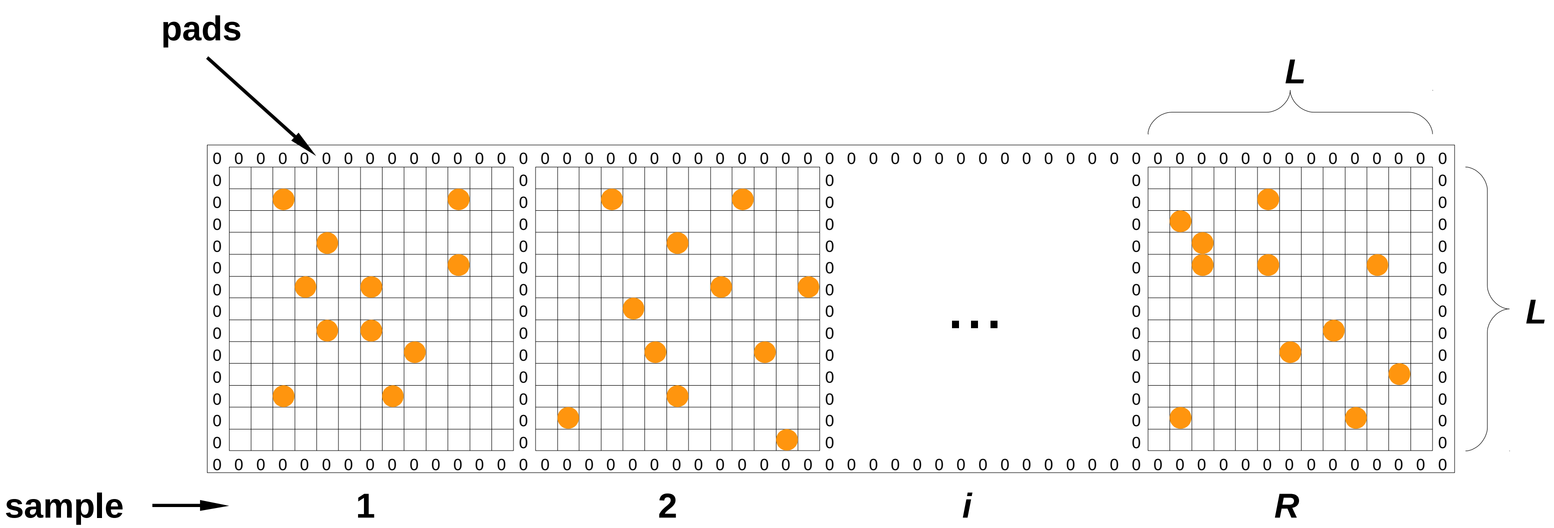
Percolation Problem

```
FOR i=1 to Number of Samples
{
    0-Generate the sample;
    1-Percolation detection algorithm;
    2-Calculate  the physics variables of interest;
    3-Calculate Averages;
}
END
```

A good simulation need up to 10^5-10^6 Number of Samples !

Multiple samples layout

Out implementation is intended to process a great number of samples simultaneously → Maximize GPU resources occupation



Series of empty border sites (or pads) are used to have open border conditions in each sample, as well as to separate independent samples during the component labeling process

Simulation in Physics

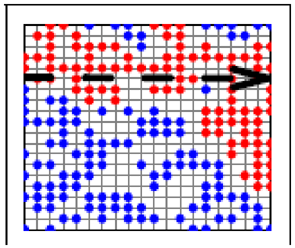
Percolation Problem using GPU and MSL

```
FOR i=1 to Number of Samples / Number of samples simultaneously simulated
{
    0-Generate the simple on gpu;
    1-Percolation detection algorithm;
    2-Calculate  the physics variables of interest;
    3-Calculate Averages;
}
END
```

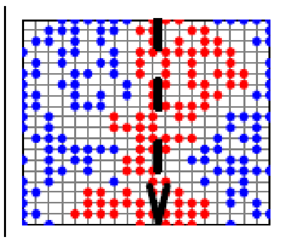
Manssen, M., Weigel, M., & Hartmann, A. K. (2012). Random number generators for massively parallel simulations on GPU. *The European Physical Journal Special Topics*, *210*(1), 53-71.

Algorithm Checking

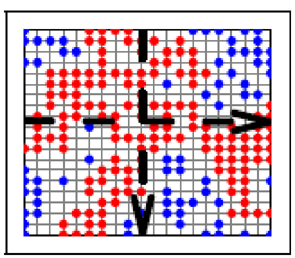
R_R percolation in x



R_D percolation in y



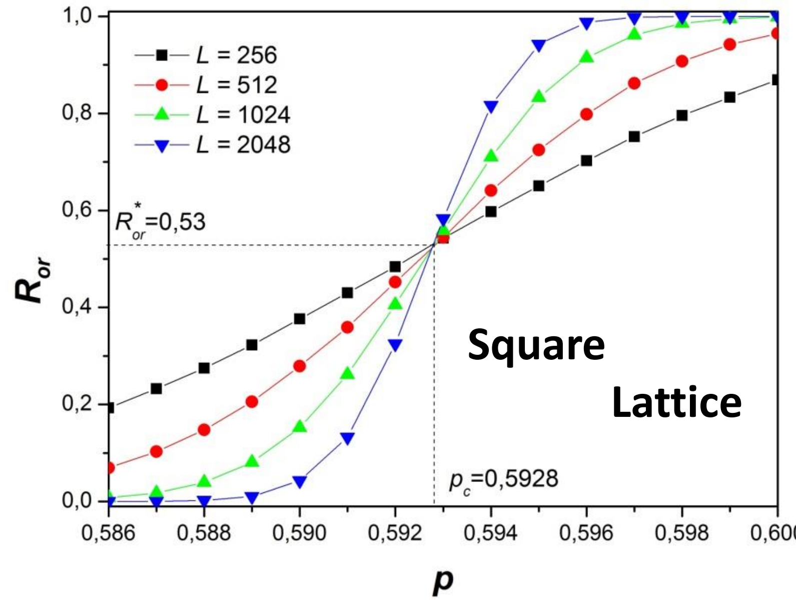
R_{or}, R_{and} percolation in x or/and y

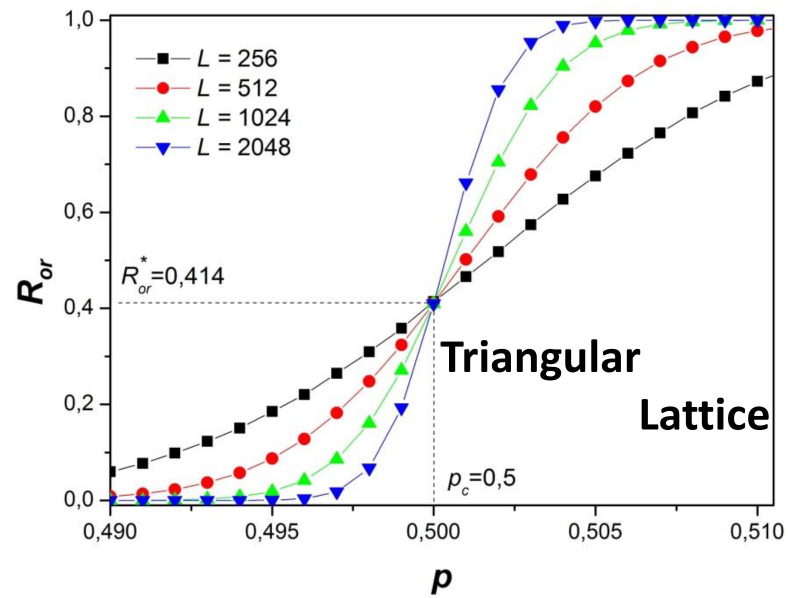


$$R_{or} = R_R \cup R_D$$

$$R_{and} = R_R \cap R_D$$

$$R_{pro} = \frac{1}{2}(R_R + R_D)$$





D. Stauffer, A. Aharony, Introduction to Percolation Theory, 2nd ed. (Taylor & Francis, London, 1994).

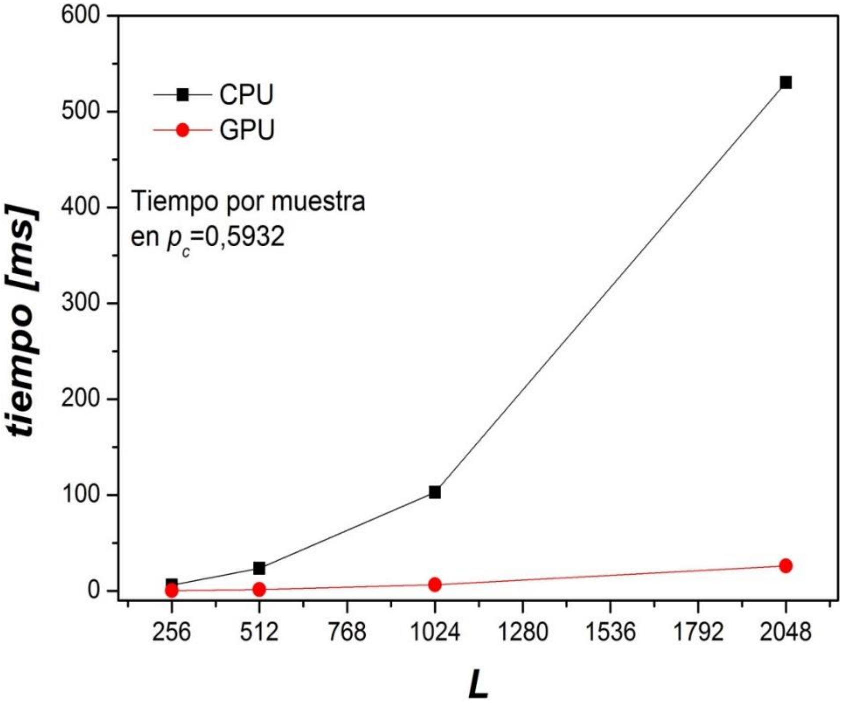
Results

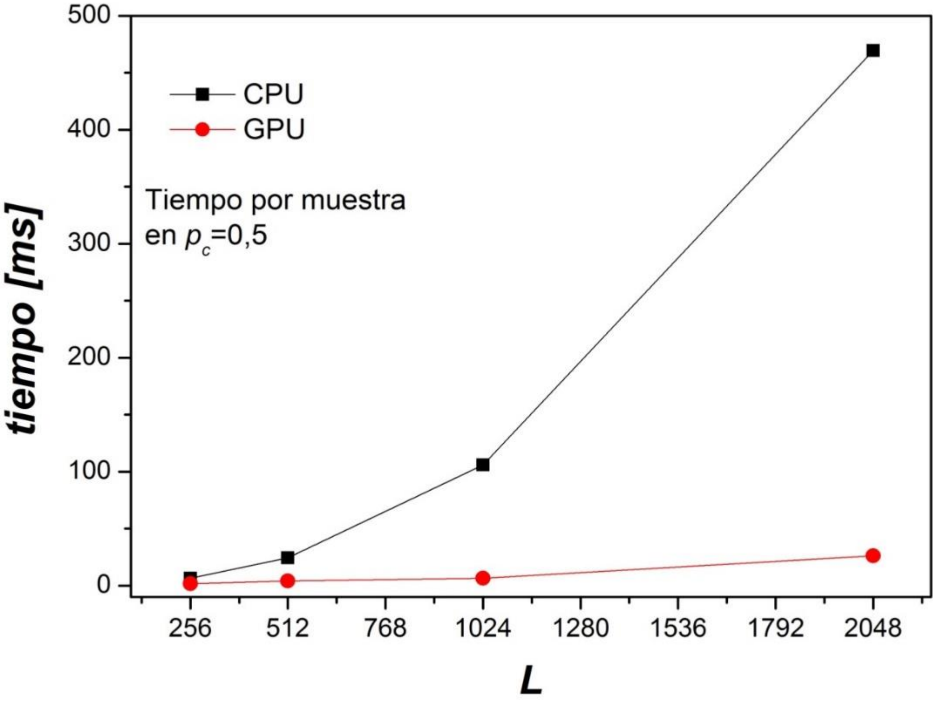
Intel i7 3770 (3.5GHz), 4GB DD3 (1333), compiled with gcc 4.1.1 (-O3 optimization).

NVIDIA GeForce GTX480 (Fermi, 480 cores), 1.5GB GDDR5, 384-bits, compiled with gcc 4.1.1 (-O3 optimization) and CUDA 4.2.

Results

With, R=15, samples simultaneously simulated





	Time(ms)			
	256×256	512×512	1024×1024	2048×2048
CPU	10.26	24.15	103.14	530.8
GPUs	8.14	2.04	7.54	26.4
Speed-up	1.26	11.84	13.68	20.11

	Time(ms)			
	256×256	512×512	1024×1024	2048×2048
CPU	7.16	25.1	106.3	469.87
GPUs	2.68	5.06	7.15	27.1
Speed-up	2.67	4.96	14.87	17.34

Conclusions

1- The use of multiple samples of simulation improves the speed up in the simulation process by a factor up to 20.

2- Still working on it ..

Thank you very much for your attention

```
__global__ void InitLabels(void)
{
        uint tid = blockIdx.x*blockDim.x + threadIdx.x;
        if( tid < (L*(R*(L+1)-1)) )
        {
                // cambio de indices de rectangulo inscripto a rectangulo completo
                uint aPos = (tid/(LRP-2))*LRP + (tid % (LRP-2)) + LRP + 1;
                uint l = Labels_d[aPos];
                // Copio Labels_d[] a CSize_d[] en parallelo...
                CSize_d[aPos] = l;
                l*=aPos;
                Labels_d[aPos] = l;

        }
}
```

```
__global__ void Scaning(void)
{
        uint tid = blockIdx.x*blockDim.x + threadIdx.x;
        if( tid < (L*(R*(L+1)-1)) )
        {
            // cambio de indices de rectangulo inscripto a rectangulo completo
                uint aPos = (tid/(LRP-2))*LRP + (tid % (LRP-2)) + LRP + 1;
                uint l = Labels_d[aPos];
                if(l)
                {
                        uint lw = Labels_d[aPos - 1];           // west
                        uint minl = UINT_MAX;
                        if(lw) minl = lw;
                        uint le = Labels_d[aPos + 1];           // east
                        if(le && (le<minl)) minl = le;
                        uint ls = Labels_d[aPos - LRP];// south
                        if(ls && (ls<minl)) minl = ls;
                        uint ln = Labels_d[aPos + LRP];         // north
                        if(ln && (ln<minl)) minl = ln;
                        // Triangular Lattice...
                        #if TRIAN==1
                        uint lnw = Labels_d[aPos + LRP - 1];    // north-west
                        if(lnw && (lnw<minl)) minl = lnw;
                        uint lse = Labels_d[aPos - LRP + 1];    // south-east
                        if(lse && (lse<minl)) minl = lse;
                        #endif
                        if(minl<l)
                        {
                                uint ll = Labels_d[l];
                                Labels_d[l] = min(ll,minl);
                                IsNotDone_d=1;
                        }
                }
        }
}
```

```
__global__ void Analysis(void)
{
		uint tid = blockIdx.x*blockDim.x + threadIdx.x;
		if( tid < (L*(R*(L+1)-1)) )
		{
			// cambio de indices de rectangulo inscripto a rectangulo completo
			uint aPos = (tid/(LRP-2))*LRP + (tid % (LRP-2)) + LRP + 1;
			uint label = Labels_d[aPos];
			if(label)
			{
				uint r=Labels_d[label];
				while(r!=label)
				{
					label = Labels_d[r];
					r = Labels_d[label];
				}
				Labels_d[aPos] = label;
			}

		}

}
```