NORGES TEKNISK-NATURVITENSKAPELIGE
UNIVERSITET

A DIVIDE AND CONQUER ALGORITHM FOR EXACT SIMULATION OF GENERAL GAUSSIAN MARKOV RANDOM FIELDS

by

Håvard Rue, Arne Marthinsen and Egil G. Husby

PREPRINT
STATISTICS NO. 11/1999

NORWEGIAN UNIVERSITY OF SCIENCE AND
TECHNOLOGY
TRONDHEIM, NORWAY

This report has URL
`http://www.math.ntnu.no/preprint/statistics/1999/S11-1999.ps`
Håvard Rue has homepage: `http://www.math.ntnu.no/~hrue`
E-mail: `hrue@math.ntnu.no`
Address: Department of Mathematical Sciences, Norwegian University of Science and Technology, N-7491
Trondheim, Norway.

A Divide and Conquer Algorithm for Exact Simulation of General Gaussian Markov Random Fields

HÅVARD RUE, ARNE MARTHINSEN AND EGIL G. HUSBY
DEPARTMENT OF MATHEMATICAL SCIENCES
NTNU, NORWAY

MARCH 12, 1999

SUMMARY

This paper presents a divide and conquer algorithm for exact simulation of general Gaussian Markov random fields. The algorithm is specially designed for large fields with a linear memory-requirement in the number of variables and has outstanding parallel properties. We report simulation experiments on a CRAY T3E distributed memory MIMD parallel computer, where we obtain a remarkable scalability of the code.

KEYWORDS: Conditional Autoregressive models, Gaussian Markov random field, Markov chain Monte Carlo, Parallel algorithm, Parallel computing.
ADDRESS FOR CORRESPONDENCE: H. Rue, Department of Mathematical Sciences, The Norwegian University of Science and Technology, N-7491 Trondheim.
E-MAIL: Havard.Rue@math.ntnu.no
ACKNOWLEDGEMENTS: This work has received support from The Research Council of Norway (Programme for Supercomputing) through a grant of computing time. HR acknowledge the EU-TMR project on Spatial Statistics (ERB-FMRX-CT960095) for support and inspiration.

1 INTRODUCTION

A Gaussian Markov Random Field (GMRF) is a Gaussian field $\boldsymbol{x}$ defined on a lattice Λ of size $n = n_1 \times n_1$. It satisfies a Markov property,

$$\pi(x_{ij} \mid x_{kl}, (k,l) \in \Lambda \backslash_{ij}) = \pi(x_{ij} \mid x_{kl}, (k,l) \in \partial_{ij}),$$

where ∂_{ij} is the neighbourhood of (i,j). Typically, the neighbourhood ∂_{ij} is a set of sites (k,l) that are close to (i,j) in some norm. We assume throughout the paper that ∂_{ij} consists of the 5×5 window centred at (i,j), but a generalisation is trivial. GMRFs are also known as conditional auto-regressions, see Besag & Kooperberg (1995) for references to GMRFs in various applications. Tjelmeland & Rue (1999) discuss how to approximate Gaussian fields on lattices with more computational convenient GMRFs. GMRFs are popular in hierarchical spatial and spatio-temporal models, see for example Wikle, Berliner & Cressie (1998).

A GMRF has the desirable property that the conditional mean and variance are given as

$$\mathrm{E}(x_{ij} \mid \boldsymbol{x}_{\partial_{ij}}) = -\sum_{(k,l)\in\partial_{ij}} \frac{Q_{ij,kl}}{Q_{ij,ij}} x_{kl} \qquad \text{and} \qquad \mathrm{Var}(x_{ij} \mid \boldsymbol{x}_{\partial_{ij}}) = 1/Q_{ij,ij}, \tag{1}$$

where the matrix $\boldsymbol{Q} = (Q_{ij,kl})$ is the precision matrix or the inverse covariance matrix $\boldsymbol{\Sigma}^{-1}$. Hence, $Q_{ij,kl}$ is zero unless x_{ij} and x_{kl} are neighbours, or more formally: x_{ij} and x_{kl} are conditionally independent given the remaining variables *iff* $Q_{ij,kl} = 0$. Thus, for a 5×5 neighbourhood, the only (possibly) non-zero elements in row (i,j) correspond to $(i-2, j-2)$, $(i-2, j-1)$, $\ldots$, $(i+2, j+2)$. This introduces a nice band-structure in $\boldsymbol{Q}$, as shown in Figure 1. Although $\boldsymbol{Q}$ is a block pentadiagonal matrix, we will refer to $\boldsymbol{Q}$ as a (general) band matrix with the implicit understanding that it is sparse. We do not need any homogeneity assumptions on the coefficients in $\boldsymbol{Q}$.

Simulating a large (general) GMRF must be done without knowing explicit the covariance matrix $\boldsymbol{\Sigma}$, because only the precision matrix $\boldsymbol{Q}$ is explicit known. However, the local conditionals (1) make it easy to derive a Gibbs-sampler algorithm (see for example Gilks, Richardson & Spiegelhalter (1996) for an introduction and further references) and run a Markov chain converging in distribution to π. The algorithm goes as follows. Start with $\boldsymbol{x} = \boldsymbol{0}$, and do until converged: choose a site at random, (i,j) say, and replace x_{ij} with a sample from the conditional univariate Gaussian with mean and variance given in (1) computed from the current $\boldsymbol{x}$. For large fields, a Gibbs-sampler algorithm is a convenient way to generate (in the limit) dependent samples from a GMRF. It is still unknown if the Markov chain has converged after N iterations although theoretical insight for GMRFs is provided by Roberts & Sahu (1997). However, if *exact* and independent samples from π can be generated within a reasonable time, it will be of great interest.

Rue (1999) notes that a GMRF can be efficiently and exactly simulated by computing the band Cholesky factorisation $\boldsymbol{Q} = \boldsymbol{L}\boldsymbol{L}^T$ and then solve by back-substitution $\boldsymbol{L}^T\boldsymbol{x} = \boldsymbol{z}$. Since $\boldsymbol{Q}$ and $\boldsymbol{L}$ are band matrices, the computations are fast, $\mathcal{O}(n^2)$ and $\mathcal{O}(n^{3/2})$ for the Cholesky factorisation and back-substitution, respectively, see Golub & van Loan (1996) for details and algorithms. However, the algorithm requires $\mathcal{O}(n^{3/2})$ bytes of memory, which is unfeasible for GMRFs on large lattices. On our computer, the algorithm is unfeasible for problems larger than about 256×256.

In this paper, we derive an alternative algorithm for large GMRFs. The algorithm is still exact, requires only $\mathcal{O}(n)$ bytes of memory and has outstanding parallel properties; we obtain speedup relatively close to the theoretical limit by investing only a modest amount of effort into the parallelization process. Large GMRFs can then be exactly simulated in reasonable time on a parallel computer. The algorithm is based on a divide and conquer idea. The numerical basis of the algorithm is fast iterative numerical methods that solve sparse positive definite linear systems. We iterate to machine precision to obtain exact samples. The serial version has arithmetic complexity of about $\mathcal{O}(n^{1.8})$, but runs slower than the algorithm in Rue (1999) for medium and small lattices.

Our algorithm can easily be extended to sample general GMRFs on non-rectangular lattices (Rue, 1999) and to do conditional simulation using Kriging (Cressie, 1993; Rue, 1999). This makes general GMRFs a class of very computationally convenient (sub-)models for spatial and spatio-temporal problems, among others.

The plan of the paper is as follows. In Section 2 we give one more result about GMRFs and introduce the reader to iterative methods for solving large sparse positive definite linear systems. We present our algorithm in Section 3, discuss implementation issues and present results on a CRAY T3E distributed memory MIMD computer in Section 4.

2 Preliminaries

2.1 Conditional Mean and Precision

We need in our algorithm the conditional mean and precision expressed in terms of the precision matrix. Split $\boldsymbol{x}$ into $\boldsymbol{x}_1$ and $\boldsymbol{x}_2$ where $\boldsymbol{x} = (\boldsymbol{x}_1^T, \boldsymbol{x}_2^T)^T$, $\boldsymbol{\mu} = (\boldsymbol{\mu}_1^T, \boldsymbol{\mu}_2^T)^T$, and divide $\boldsymbol{\Sigma}$ and $\boldsymbol{Q}$ into appropriate blocks

$$\boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma}_{11} & \boldsymbol{\Sigma}_{12} \\ \boldsymbol{\Sigma}_{21} & \boldsymbol{\Sigma}_{22} \end{bmatrix} \quad \text{and} \quad \boldsymbol{Q} = \begin{bmatrix} \boldsymbol{Q}_{11} & \boldsymbol{Q}_{12} \\ \boldsymbol{Q}_{21} & \boldsymbol{Q}_{22} \end{bmatrix}.$$

The natural relation between conditional independence and the precision matrix becomes clear when we restate the classical result about the conditional density $\boldsymbol{x}_1 \mid \boldsymbol{x}_2$, in terms of the $\boldsymbol{Q}_{ij}$'s instead of the $\boldsymbol{\Sigma}_{ij}$'s. The following result seems to be less well known than it ought to be.

THEOREM 1 *The conditional density for $\boldsymbol{x}_1 \mid \boldsymbol{x}_2$ is Gaussian with mean $\boldsymbol{\mu}_{1|2} = \boldsymbol{\mu}_1 - \boldsymbol{Q}_{11}^{-1}\boldsymbol{Q}_{12}(\boldsymbol{x}_2 - \boldsymbol{\mu}_2)$ and precision $\boldsymbol{Q}_{1|2} = \boldsymbol{Q}_{11}$.*

The proof is simply to use the block equations derived from $\boldsymbol{Q\Sigma} = \boldsymbol{I}$ and the standard results about conditional mean and variance.

2.2 Solving Large Positive Definite Linear Systems

In Section 3 it will be clear that one of the main ingredients in our algorithm is the solution of large systems of linear equations $\boldsymbol{Qx} = \boldsymbol{b}$, where $\boldsymbol{Q}$ is a $n \times n$ precision matrix with band-structure shown in Figure 1. We will use iterative methods for sparse, symmetric, positive definite systems since $\boldsymbol{Q}$, by definition, has these properties. As noted in Section 1, a direct solution in terms of Cholesky factorisation followed by forward elimination and back-substitution has a memory requirement that is $\mathcal{O}(n^{3/2})$, which is unacceptable for large systems. Iterative techniques need only $\mathcal{O}(n)$ memory.

A number of (iterative) methods implement certain (orthogonal) projection techniques onto subspaces of the solution space. Methods like steepest descent, GMRES and the conjugate gradient algorithms belong to this class. For the conjugate gradient method, the jth search direction is in the degree-j Krylov space of $\boldsymbol{Q}$ at the initial residual $\boldsymbol{r}_0$. The algorithm can be written as follows, where we interpret $\boldsymbol{M}$ as the identity-matrix for the moment:

compute $\boldsymbol{r}_0 = \boldsymbol{b} - \boldsymbol{Q}\boldsymbol{x}_0$, $\boldsymbol{z}_0 = \boldsymbol{M}^{-1}\boldsymbol{r}_0$ and $\boldsymbol{p}_0 = \boldsymbol{z}_0$
for $j = 0, 1, \dots$, until convergence do

$$
\begin{aligned}
\boldsymbol{\alpha}_j &= \boldsymbol{r}_j^T \boldsymbol{z}_j / \boldsymbol{p}_j^T \boldsymbol{Q} \boldsymbol{p}_j \\
\boldsymbol{x}_{j+1} &= \boldsymbol{x}_j + \boldsymbol{\alpha}_j \boldsymbol{p}_j \\
\boldsymbol{r}_{j+1} &= \boldsymbol{r}_j - \boldsymbol{\alpha}_j \boldsymbol{Q} \boldsymbol{p}_j \\
\boldsymbol{z}_{j+1} &= \boldsymbol{M}^{-1} \boldsymbol{r}_{j+1} \\
\boldsymbol{\beta}_j &= \boldsymbol{r}_{j+1}^T \boldsymbol{z}_{j+1} / \boldsymbol{r}_j^T \boldsymbol{z}_j \\
\boldsymbol{p}_{j+1} &= \boldsymbol{z}_{j+1} + \boldsymbol{\beta}_j \boldsymbol{p}_j
\end{aligned}
$$

end

We monitor convergence of the above algorithm by computing some norm of the residual vector $\boldsymbol{z}_{j+1}$. We stop the iterations when we reach the machine precision and then the solution is $\boldsymbol{x}_{j+1}$. Note that the algorithm only involves one matrix–vector multiplication and three inner products as well as linear combinations of vectors in each iteration, all fast operations. We have to store the matrix $\boldsymbol{Q}$ and four additional vectors, hence the memory requirement is $\mathcal{O}(n)$. Implementation issues of the conjugate gradient method are discussed in Golub & van Loan (1996), Saad (1996) and Velde (1994).

If $\boldsymbol{Q}$ has full rank, the algorithm converges after at most n iterations in exact arithmetic. However, the accuracy of the iterates is often much better than one might expect from convergence theorems, and the iteration can be terminated with $j \ll n$. The convergence of the conjugate gradient method is very fast in the $\boldsymbol{Q}$-norm $\|\boldsymbol{w}\|_{\boldsymbol{Q}}^2 = \boldsymbol{w}^T\boldsymbol{Q}\boldsymbol{w}$ if the condition number of $\boldsymbol{Q}$ is close to one. In general, the convergence of the iterative techniques depends on the condition number of the coefficient matrix

$\boldsymbol{Q}$, and increasing condition number leads usually to more iterations.

It is common to improve the convergence-rate of the conjugate gradient algorithm by preconditioning. Preconditioning techniques involve finding a symmetric and positive definite matrix $\boldsymbol{M}$ such that $\boldsymbol{M}^{-1}\boldsymbol{Q}$ is better conditioned (in the $\boldsymbol{Q}$-norm) than the original matrix $\boldsymbol{Q}$. $\boldsymbol{M}$ should hence approximate $\boldsymbol{Q}$ and be easily invertible. So, instead of solving the original linear system, we solve $\widetilde{\boldsymbol{Q}}\boldsymbol{x} = \widetilde{\boldsymbol{b}}$, where $\widetilde{\boldsymbol{Q}} = \boldsymbol{M}^{-1}\boldsymbol{Q}$ and $\widetilde{\boldsymbol{b}} = \boldsymbol{M}^{-1}\boldsymbol{b}$.

There exists a large number of preconditioners (Golub & van Loan, 1996; Oppe, Joubert & Kincaid, 1988), and we have chosen to use *modified incomplete Cholesky preconditioning* with zero fill-in. Incomplete Cholesky factorisation of the original matrix $\boldsymbol{Q}$ gives $\boldsymbol{Q} = \boldsymbol{M} - \boldsymbol{R} = \widehat{\boldsymbol{L}}\widehat{\boldsymbol{L}}^T - \boldsymbol{R}$, where $\boldsymbol{R}$ is the error of the factorisation and $\widehat{\boldsymbol{L}}$ has the same zero-pattern as the lower part of the $\boldsymbol{Q}$ matrix. *Modified* incomplete Cholesky factorisation attempts to reduce the effects from requiring zero fill-in. This is done by adjusting the diagonal entries in $\widehat{\boldsymbol{L}}$ so that $\boldsymbol{Q} - \widehat{\boldsymbol{L}}\widehat{\boldsymbol{L}}^T$ has zero row sums. For many matrices, this requirement gives a better condition number for $\boldsymbol{M}^{-1}\boldsymbol{Q}$ than for the non-modified incomplete Cholesky preconditioning. Note that solving $\boldsymbol{z} = \boldsymbol{M}^{-1}\boldsymbol{r}$ is done by one forward and one back-substitution, $\widehat{\boldsymbol{L}}\boldsymbol{u} = \boldsymbol{r}$ and $\widehat{\boldsymbol{L}}^T\boldsymbol{z} = \boldsymbol{u}$. Further, when we solve $\widetilde{\boldsymbol{Q}}\boldsymbol{x} = \widetilde{\boldsymbol{b}}$ for many right hand sides, i.e. compute $\widetilde{\boldsymbol{Q}}^{-1}\boldsymbol{B}$ for a matrix $\boldsymbol{B}$, we only need to compute the preconditioning matrix once.

We have in our implementation (see Section 4.2) used a sequential implementation of the preconditioned conjugated gradient method from the software library NSPCG (Oppe et al., 1988). This is a package with routines that solve large, sparse systems of linear equations using iterative methods.

3 THE ALGORITHM

Based on the preliminary results in Section 2, we can now derive our divide and conquer algorithm. We present the basic divide and conquer step in Section 3.1 and the full algorithm in Section 3.2. Define $\boldsymbol{x}_i$ as the ith column of $\boldsymbol{x}$ with mean $\boldsymbol{\mu}_i$, $\boldsymbol{\Sigma}_{ij}$ and $\boldsymbol{Q}_{ij}$ as the covariance and precision between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$.

3.1 THE DIVIDE AND CONQUER STEP

Figure 2 illustrates our divide and conquer approach to simulate a GMRF. We do successive division of the lattice starting with the two middle columns c and $c+1$, where c is the integer value of $n_1/2$ when n_1 is odd. Let $\boldsymbol{x}_s = (\boldsymbol{x}_c^T, \boldsymbol{x}_{c+1}^T)^T$ denote the two columns. Define $\boldsymbol{x}_A = (\boldsymbol{x}_1^T, \dots, \boldsymbol{x}_{c-1}^T)^T$ and $\boldsymbol{x}_B = (\boldsymbol{x}_{c+2}^T, \dots, \boldsymbol{x}_n^T)^T$, so that $\boldsymbol{x} = (\boldsymbol{x}_A^T, \boldsymbol{x}_s^T, \boldsymbol{x}_B^T)^T$ and $\boldsymbol{\mu} = (\boldsymbol{\mu}_A^T, \boldsymbol{\mu}_s^T, \boldsymbol{\mu}_B^T)^T$. We split $\boldsymbol{Q}$ similarly

$$\boldsymbol{Q} = \begin{bmatrix} \boldsymbol{Q}_{AA} & \boldsymbol{Q}_{As} & \boldsymbol{0} \\ \boldsymbol{Q}_{sA} & \boldsymbol{Q}_{ss} & \boldsymbol{Q}_{sB} \\ \boldsymbol{0} & \boldsymbol{Q}_{Bs} & \boldsymbol{Q}_{ss} \end{bmatrix}.$$

Note that $\boldsymbol{Q}_{AB}$ is zero since $\boldsymbol{x}_s$ is a separating subset for $\boldsymbol{x}_A$ and $\boldsymbol{x}_B$, $\boldsymbol{x}_A \perp \boldsymbol{x}_B \mid \boldsymbol{x}_s$. Assume for a moment that we can simulate $\boldsymbol{x}_s$ from its marginal density $\pi(\boldsymbol{x}_s)$. We will later show how to compute $\pi(\boldsymbol{x}_s)$ efficiently. A sample from the joint density $\pi(\boldsymbol{x})$ is then found by sampling $\boldsymbol{x}_A, \boldsymbol{x}_B \mid \boldsymbol{x}_s$ from the conditional density in Theorem 1. The precision matrix and mean for $\boldsymbol{x}_A, \boldsymbol{x}_B \mid \boldsymbol{x}_s$ is

$$\boldsymbol{Q}_{AB|s} = \begin{bmatrix} \boldsymbol{Q}_{AA} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Q}_{BB} \end{bmatrix} \tag{2}$$

and

$$\boldsymbol{\mu}_{AB|s} = \begin{bmatrix} \boldsymbol{\mu}_{A|s} \\ \boldsymbol{\mu}_{B|s} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\mu}_{A} \\ \boldsymbol{\mu}_{B} \end{bmatrix} - \begin{bmatrix} \boldsymbol{Q}_{AA}^{-1} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{Q}_{BB}^{-1} \end{bmatrix} \begin{bmatrix} \boldsymbol{Q}_{As} \\ \boldsymbol{Q}_{Bs} \end{bmatrix} (\boldsymbol{x}_s - \boldsymbol{\mu}_s). \tag{3}$$

The important observations are the following.

- The conditional precisions equal $\boldsymbol{Q}_{AA}$ and $\boldsymbol{Q}_{BB}$ which are known already as sub-blocks of $\boldsymbol{Q}$. Further, $\boldsymbol{Q}_{AA}$ and $\boldsymbol{Q}_{BB}$ have the same band structure as $\boldsymbol{Q}$.
- The linear equation for the conditional mean (3) splits into two separate equations, one for $\boldsymbol{\mu}_{A|s}$ and one for $\boldsymbol{\mu}_{B|s}$, since $\boldsymbol{Q}_{AB} = \boldsymbol{0}$.
- There is no need the invert $\boldsymbol{Q}_{AA}$ and $\boldsymbol{Q}_{BB}$ as we only need to compute $\boldsymbol{Q}_{AA}^{-1}$ times a vector, $\boldsymbol{y} = \boldsymbol{Q}_{AA}^{-1}\boldsymbol{z}$, say. This is done by solving the linear system $\boldsymbol{Q}_{AA}\boldsymbol{y} = \boldsymbol{z}$, see Section 2.2.

Equations (2) and (3) show that *if* we are able to sample $\boldsymbol{x}_s$ fast, then we can compute the conditional mean and precision for $\boldsymbol{x}_{A|s}$ and $\boldsymbol{x}_{B|s}$ by solving two band linear systems. Since both $\boldsymbol{Q}_{A|s}$ and $\boldsymbol{Q}_{B|s}$ have the same band structure as $\boldsymbol{Q}$, we have now divided our problem into two separate, smaller ones.

The next task is to compute the marginal density for $\boldsymbol{x}_s$. Since only the precision matrix $\boldsymbol{Q}$ is available, we need to compute the covariance matrix $\boldsymbol{\Sigma}_{ss}$ for $\boldsymbol{x}_s$. The idea is as follows: Since $\boldsymbol{Q\Sigma} = \boldsymbol{I}$, we can write out the equations for block-column s in $\boldsymbol{\Sigma}$

$$\boldsymbol{Q} \begin{bmatrix} \boldsymbol{\Sigma}_{1s} \\ \vdots \\ \boldsymbol{\Sigma}_{ss} \\ \vdots \\ \boldsymbol{\Sigma}_{n_1 s} \end{bmatrix} = \begin{bmatrix} \boldsymbol{0} \\ \vdots \\ \boldsymbol{I} \\ \vdots \\ \boldsymbol{0} \end{bmatrix}. \tag{4}$$

The right hand side is zero except for a $2n_1 \times 2n_1$ identity matrix $\boldsymbol{I}$. Note that (4) defines $2n_1$ band linear systems of size $n \times n$. Hence, for the kth solution $\boldsymbol{y}^{(k)}$, column k of $\boldsymbol{\Sigma}_{ss}$ is available as

$$(\boldsymbol{\Sigma}_{ss})_k = (y^{(k)}_{(c-1)n_1+1}, \cdots, y^{(k)}_{(c-1)n_1+2n_1})^T.$$

We can then simulate $\boldsymbol{x}_s$ by computing the Cholesky decomposition $\boldsymbol{\Sigma}_{ss} = \boldsymbol{LL}^T$ at the cost $\mathcal{O}(n\sqrt{n})$ flops, and then use the identity $\boldsymbol{x}_s \stackrel{d}{=} \boldsymbol{\mu}_s + \boldsymbol{Lz}$, where $\boldsymbol{z}$ is a $2n_1 \times 1$ vector of iid standard Gaussian variates.

3.2 The Full Algorithm

We are now able to present our full algorithm based on the basic divide and conquer step. It goes as follows:

1. Solve the $2n_1$ band linear equation systems in (4) to obtain $\boldsymbol{\Sigma}_{ss}$.
2. Compute the Cholesky decomposition $\boldsymbol{\Sigma}_{ss} = \boldsymbol{L}\boldsymbol{L}^T$.
3. Compute $\boldsymbol{x}_s = \boldsymbol{\mu}_s + \boldsymbol{L}\boldsymbol{z}$, where $\boldsymbol{z}$ is a vector of iid standard Gaussian variables.
4. Solve the two band linear systems in (3) to obtain $\boldsymbol{\mu}_{A|s}$ and $\boldsymbol{\mu}_{B|s}$.

Our $n_1 \times n_1$ problem is now divided into two separate problems of size $n_1 \times c - 1$ and $n_1 \times n_1 - c - 1$, of *the same form* as our original one. Note that all dependency of $\boldsymbol{x}_s$ is included in $\boldsymbol{\mu}_{A|s}$ and $\boldsymbol{\mu}_{A|s}$, so that $\boldsymbol{x}_s$ is not used later in the algorithm. We can therefore use our algorithm again at each of the two separate problems and proceed similarly by each time dividing the largest size (either a row or a column) until either the size is so small that it can not be divided into two smaller ones (i.e. 1×1, 1×2, ..., 3×3), or it is more efficient to do a direct simulation: compute the covariance matrix, the Cholesky decomposition and then simulate using the classic procedure. Figure 2 illustrates the first three steps in the algorithm. After we have finished the A-part in Figure 2, we continue with the B-part in the same manner. Thus our algorithm is complete.

Steps 1–4 are needed for the first sample, while repeated samples need only step 3 and 4 (we store all Cholesky decompositions). Thus, producing many iid samples from π is particularly efficient with our algorithm.

4 Parallel Computing

Parallel computing have become the state of art within high performance computing. Only a limited number of applications are able to take full advantage of massively parallel systems, but even with a very limited effort most codes can efficiently use modest number of processors (e.g. in the range 4 – 64 processors).

The two main categories of parallel computers are *distributed* and *shared* memory systems. Without taking into account the particular application at hand, it is not immediately clear from what category to choose the computer. Our choice has been a distributed memory MIMD computer, the CRAY T3E with DEC Alpha EV5 RISC processors, using explicit message passing between the processors.

Each processing unit of a distributed memory MIMD computer can be thought of as a separate computer. The units are interconnected with a fast network. The tasks that together implement the algorithm are divided onto the available processors. When data in the memory on one processor is needed on another, explicit communication (`send` and `receive`) must take place. A number of communication libraries exist. In our simulations we have used MPI (Foster, 1995).

When implementing a parallel simulation algorithm, the data distribution strategy will have a large impact on the performance of the code. For an algorithm to perform efficiently, data must be available at each processor when needed and the required communication should be minimised. There are many reasons why a parallel simulation algorithm should have data distributed in a static data structure across the processors throughout the whole simulation. We mention in particular that gathering and scattering of data are operations that can be regarded as sequential operations since they do not advance the computations. The sum of their time consumption in each pass through the loop will often be significant compared to the total simulation time. We also emphasise that collection of data onto one single processor severely limits the scalability of the code. By using a static, distributed data-structure, we get access to a (scalable) larger amount of memory.

4.1 Parallelisation of the simulation algorithm

It is clear that complete parallelisation of our simulation algorithm is a major task. Since our algorithm only requires $\mathcal{O}(n)$ memory, we will focus merely on the time usage of the simulation algorithm. Our algorithm has the rare property, that with *minimal* effort parallelising the code, we obtain a speedup close to the theoretical limit! We consider first parallelisation of the algorithm when it is run only for one sample, then we discussed parallelisation for repeated samples.

Assume for the moment that we want only one sample from the GMRF. Instrumentation of the sequential code revealed that, for a 100×100 system, more than 99% of the total time was spent solving the linear systems in (4). For larger systems, this percentage increases. Note that (4) is a perfectly parallel operation, since each column of $\mathbf{\Sigma}_{ss}(=\texttt{covmat})$ can be computed independently of the others. The loop performing this operation in the code is

```
for k=0 to size-1 do
   bvec[:]=0.0;         // the right hand side vector is zero,
   bvec[start+k]=1.0;   // except for the one entry which is 1.0

                        // solve Qx=bvec. we compute the
                        // preconditioner only the first
                        // time (specified by the fourth argument)
   solve_pcg(Q,bvec,precond,(k==0?1:0));

   for i=0 to size-1 do // store the vector x in column k in covmat
      covmat[k*size+i]=bvec[start+i];
   end;
end;
```

To parallelize this critical part of the code, imagine that `nproc` processors are running simultaneously and that the variable me (defined by the computer) uniquely identifies each processor ($0 \leq$ me $<$ nproc). All processors execute the same code, and we can control what each processor should do merely by including the identifyers me and nproc into the control structures in the code. We let each processor solve a certain fraction of the systems, and, after all processors have finished their loops, we collect their partial answers into covmat. The following code does exactly this.

```
n1=size/nproc;           // integer division
n2=size%nproc;           // remainder division
n3=n1+(me<n2?1:0);       // number of columns for processor ''me''
n4=me*n1+(me<n2?me:n2); // first column for processor ''me''

for k=n4 to n3+n4-1
   bvec[:]=0.0;
   bvec[start+k]=1.0;

   solve_pcg(Q,bvec,precond,(k==n4?1:0));

   for i=0 to size-1 do // store the vector x in column k in covmat
      covmat[k*size+i]=bvec[start+i];
   end;
end;
                         // add contributions from the different processors
global_sum(sqr(size),nproc,covmat);
```

We see that the only essential differences between these two codes are the lower and upper bound for the loop and that a global sum is computed when the latter loop has finished. If $n/\texttt{nproc}$ is an integer, all processors compute the same number of columns in $\mathbf{\Sigma}_{ss}$. If not, we let the first $n \bmod \texttt{nproc}$ processors compute $\lceil n/\texttt{nproc} \rceil$ columns. Since all columns are of equal length, we are ensured load balance among the processors. Prior to entering the loop, we initialize all entries in covmat with zero. Each processor fills one rectangular part of the array, and when the loop is completed we sum covmat across all the processors. The result is the marginal covariance matrix $\mathbf{\Sigma}_{ss}$.

The remaining part of the algorithm runs sequentially, also the solution of the two linear systems (3). Of course, we could parallelize also this part. It whould have been a major task compared to parallelisation of the above loop, and the marginal gain obtained by doing this will not justify the extra effort.

The picture changes somewhat if we apply our algorithm to sample n_s iid samples from the GMRF. Instead of dividing the algorithm into two parts as mentioned in Section 3.2 (steps 1–4 for the first sample, then steps 3 and 4 for each repeated one) we chose another strategy in the parallel implementation. It is both easier and more efficient to sample all n_s samples simultaneously (or if n_s is very large we do only a fraction at the time). Recall the solution of the two linear systems (3) is done sequentially if $n_s = 1$ and this is how $\boldsymbol{x}_s$ propagates to other parts of the lattice. Hence for nproc processors we can compute $\texttt{nproc} - 1$ extra samples completely for free, and the remaining nearly for free (see Section 3.2).

4.2 Simulation Experiments on a CRAY T3E

We will report some simulation experiments on the CRAY T3E. The speed-up is defined as ratio of the time using one and `nproc` processors on one particular computer. The perfect speed-up is `nproc`. We report results for sampling $n_s = 1$ sample for a 100×100 and a 300×300 lattice. From the discussion in Section 4.2 it is clear that this yields a *conservative* picture of the speed-up: the gain increases as n_s increases, and it also increases as the size of the lattice increases.

We use a 5×5 neighbourhood and the coefficients in Table 1 to fit a stationary Gaussian field with correlation function $\exp(-3(d/20)^2)$, where d is the distance in pixels, see Tjelmeland & Rue (1999) for details and further discussion along these lines. Figure 3 shows an exact sample from the GMRF.

Figure 4 shows the simulation results. The upper left and upper right graphs show the elapsed time for a 100×100 and a 300×300 lattice, respectively, as a function of the number of processors. (We had only access to 24 processors simultaneously.) The lower left and lower right graphs show the speed-up and the efficiency, the latter defined as the ratio between the obtained speed-up and the number of processors we use.

The obtained speed-up is very good and close to the theoretical limit. In most cases the gain will even better: increasing the size of the lattice will increase the speed-up. Further, we obtain $n_s = \texttt{nproc}$ samples for the same cost as one sample, a fact we have *not* accounted for in the speed-up. There is nothing indicating, neither empirically nor theoretically, that the speed-up should be less good for more than 24 processors for large fields.

To investigate the arithmetic complexity of our algorithm, we ran a sequential version for various coefficients and neighborhoods, and found it to be about $\mathcal{O}(n^{1.8})$.

REFERENCES

BESAG, J. & KOOPERBERG, C. (1995). On conditional and intrinsic autoregression, *Biometrika* 82(4): 733–746.

CRESSIE, N. A. C. (1993). *Statistics for spatial data*, 2 edn, John Wiley, New York.

FOSTER, I. (1995). *Designing and Building Parallel Programs*, Addison–Wesley Publishing Company.

GILKS, W. R., RICHARDSON, S. & SPIEGELHALTER, D. J. (1996). *Markov Chain Monte Carlo in Practice*, London: Chapman & Hall.

GOLUB, G. H. & VAN LOAN, C. F. (1996). *Matrix Computations*, 3 edn, Johns Hopkins University Press, Baltimore.

OPPE, T. C., JOUBERT, W. D. & KINCAID, D. R. (1988). *NSPCG User's Guide Version 1.0 — A Package for Solving Large Sparse Linear Systems by Various Iterative Methods*, Center for Numerical Analysis, The University of Texas at Austin.

ROBERTS, G. O. & SAHU, S. K. (1997). Updating schemes, correlation structure, blocking and parameterization for the Gibbs sampler, *Journal of the Royal Statistical Society, Series B* 59(2): 291–317.

RUE, H. (1999). A fast and exact simulation algorithm for general Gaussian Markov random fields, *Statistics no. 8*, Department of Mathematical Sciences, Norwegian University of Science and Technology, Trondheim, Norway.

SAAD, Y. (1996). *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company.

TJELMELAND, H. & RUE, H. (1999). Approximating Gaussian fields with Gaussian Markov random fields, *in progress*, Department of Mathematical Sciences, Norwegian University of Science and Technology, Trondheim, Norway.

VELDE, E. F. V. D. (1994). *Concurrent Scientific Computing*, number 16 in *TAM*, Springer–Verlag.

WIKLE, C. K., BERLINER, L. M. & CRESSIE, N. A. (1998). Hierarchical Bayesian space-time models, *Environmental and Ecological Statistics* 5(2): 117–154.

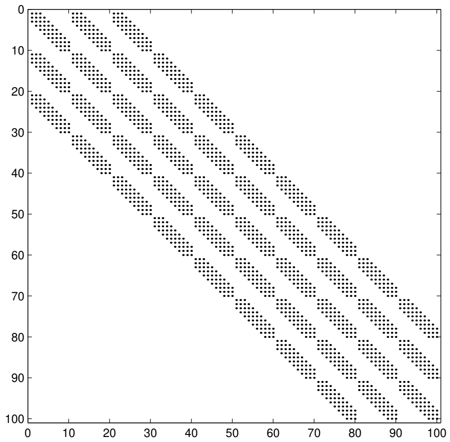

FIGURE 1: The figure displays the (possible) non-zero elements in the $\boldsymbol{Q}$ matrix for a 10×10 lattice and a 5×5 neighbourhood.

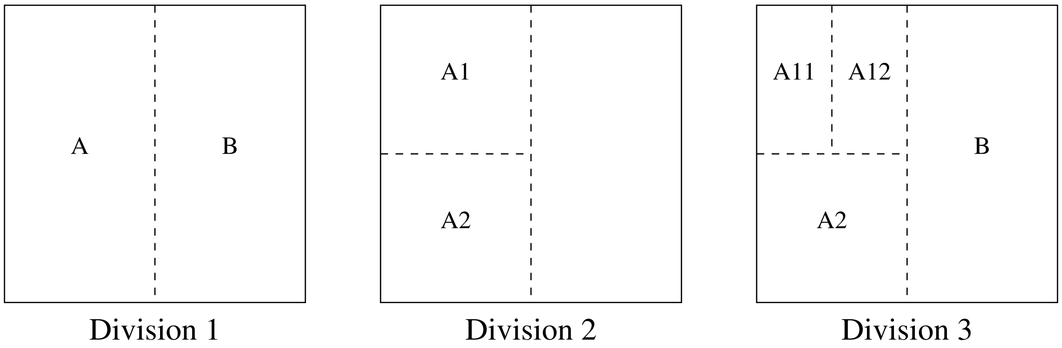

FIGURE 2: The three first step in the algorithm.

	1.721	4.085	0.061
y	-16.954	-15.555	4.085
	90.231	-16.954	1.721
		x	

TABLE 1: The upper right part of the 5×5 neighbourhood used in the presented simulation results, where the remaining coefficients are found by symmetry. The coefficients are found by approximating the GMRF to a Gaussian field with correlation function $\exp(-3(d/20)^2)$, where d is the distance in pixels, see Tjelmeland & Rue (1999) for details.

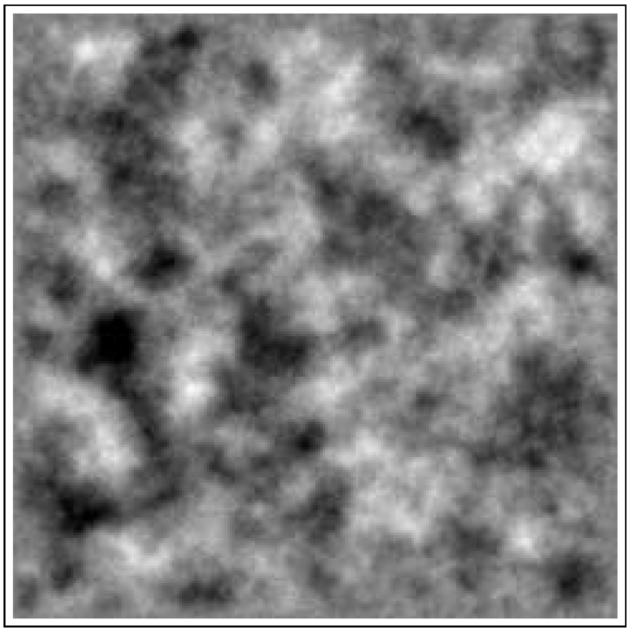

FIGURE 3: An exact sample from a GMRF with the neighbourhood given in Table 1.

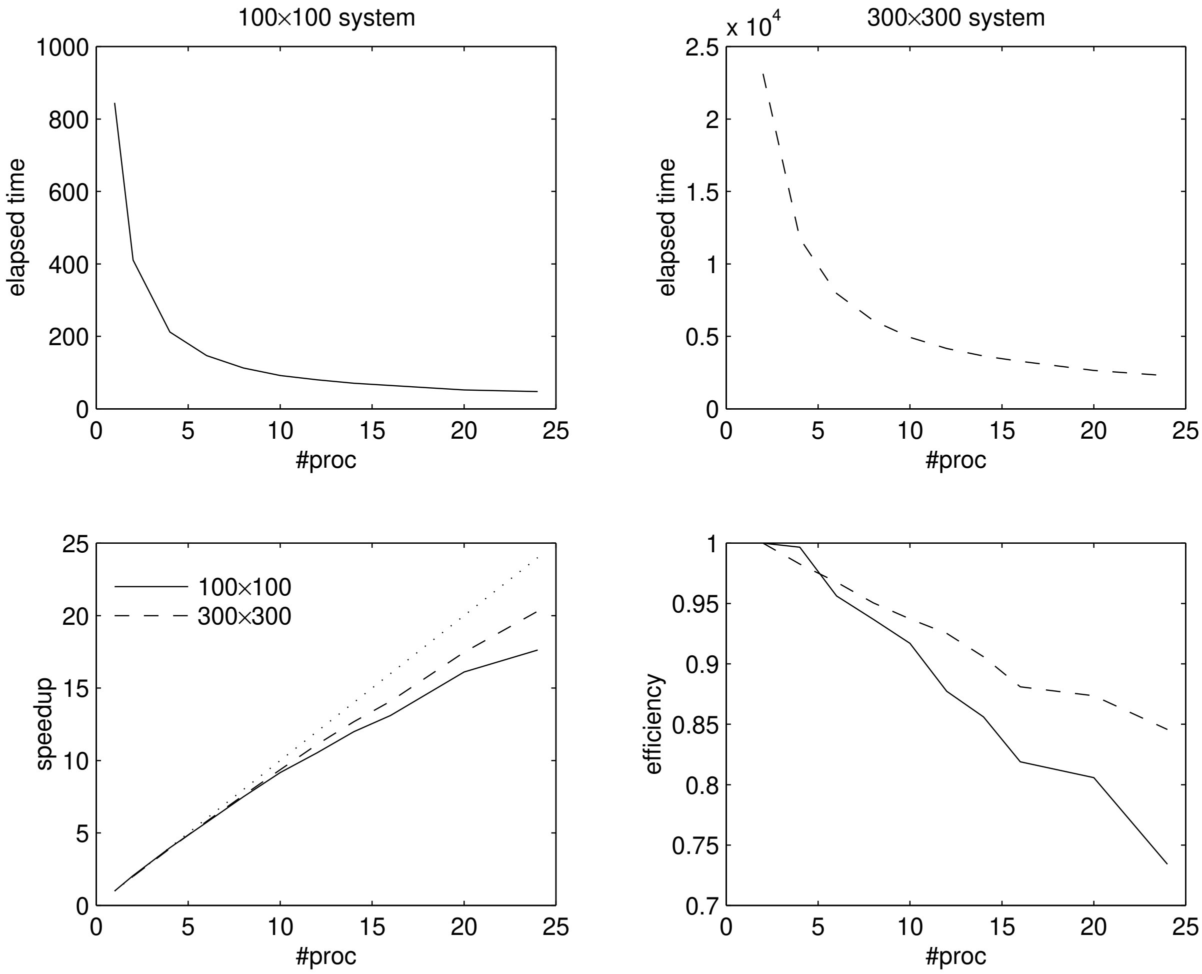

FIGURE 4: Elapsed time and speedup versus number of processors for iterative solution of a 100×100 and a 300×300 system. The dotted line in the lower left figure indicates perfect speedup. The legend is valid for all the plots.