SKIRON

The Weather Forecasting System SKIRON

Volume IV

PARALLELIZATION OF THE MODEL

SKIRON

Athens

June 1998

SKIRON

The Weather Forecasting System SKIRON

Volume IV

PARALLELIZATION OF THE MODEL

by

N.M. Missirlis, L.A. Boukas and N.Th. Mimikou

Department of Informatics, Section of Theoretical Informatics, University of Athens

SKIRON

Athens

June 1998

About this documentation

The **SKIRON** documentation consists of six volumes namely :

Vol. I	***Preprocessing***
Vol. II	***Description of the model***
Vol. III	***Numerical methods***
Vol. IV	***Parallelization of the model***
Vol. V	***Postprocessing and graphics***
Vol. VI	***Installation - Operation guides***

SKIRON

Copies of these volumes are available from Dr. G. Kallos, Univercity of Athens, Department of Applied Physics, Panepistimioupolis Bldg PHYS-5, 17584 Athens, Greece, email: Kallos@etesian.dap.uoa.gr

ISBN No.

SET : 960-8468-14-0

VOL IV : 960-8468-18-3

Acknowledgments

This work has been performed at the framework of the SKIRON project (EPET #322) which has been founded by the Greek Government (through the General Secretariat of Science and Technology) and European Union.

The scope of the SKIRON project was to develop a regional weather forecasting system on parallel computer platforms.

For the purpose of this project, a consortium has been established between the University of Athens (Dept. of Applied Physics and Dept. of Informatics), the Hellenic National Meteorological Service, the Brainware S.A., the Athens High Performance Computer Laboratory and the Innovative Technologies Center S.A. Coordinator of the project was Dr. G. Kallos of the University of Athens.

We would like to thank all the colleagues and institutions for their help and support in preparing this documentation. Especially, we would like to thank Z.I.Janjic, F. Mesinger and T.Black for comments and suggestions. Thanks to the scientists of the Hydrometeorological Institute for their assistance. Special thanks to the members of the Atmospheric Modeling Group of the University of Athens, S. Nickovic, A. Papadopoulos, V.Kotroni, K. Lagouvardos, D. Jovic, O. Kakaliagou, M. Varinou, P. Katsafados, V. Kojanic and D. Nickovic for their suggestions, corrections and proof writing. Also, special thanks to the members of the Parallel Scientific Computing Group of the Department of Informatics, Univercity of Athens, Y. Kotronis, G. Kounias, N. Argyropoulos, F. Tjaferis, K. Moutselos, D. Nikolaidis and to D. Dimitrelos of the Athens High Performance Computer Laboratory, for their help to the Volume IV.

Acknowledgment is made to the US National Center for Atmospheric Research (NCAR) which is sponsored by the National Science Foundation for making available to use the Convex SPP2000 machine.

SKIRON

VOLUME IV

TABLE OF CONTENTS

1.The SPP-1600 parallel computer 1

1.1.Introduction 1
1.2.Architecture of the SPP-1600 1
1.3.Subcomplexes.. 2
1.4.SPP Features 2

2.Parallel programming models 3

2.1.Introduction 3
2.2.Message Passing.. 4

3.The PVM (Parallel Virtual Machine) programming environment.......... 6

3.1 History of PVM.. 6
3.2.PVM/GSM (Parallel Virtual Machine for Globally Shared Memory) 7
3.3.Capabilities... 7
3.4.PVM Applications 8

4.The MPI (Message Passing Interface) programming environment........ 9

4.1.History of MPI 9
4.2.MPICH/GSM (Message Passing Interface for Globally Shared Memory) 9
4.3.Capabilities... 10

5.The Eta model 11

5.1.Parallelization of the Eta model 12
5.1.1.Domain Decomposition 13

SKIRON

5.2.Parallel Implementation ... 18
5.2.1.The parallelization principles and techniques ... 18
5.2.2.Determination of kernel and Halo points ... 20
5.2.3.Communication ... 24
5.2.4.Output Data Acquisition ... 29
5.3.Numerical Results ... 30

6.Introduction to the parallel code ... 35

6.1.Creation of the input files ... 35
6.2.Creation of the input files relative to the boundary conditions ... 36
6.3.Storage of significant values ... 38
6.4.Initialization of the parallel program ... 39
6.5.Creation of the two dimensional mesh topology ... 40
6.6.Termination of the parallel program ... 41
6.7.Local communication ... 41
6.8.Global Communication ... 44
6.9.Mixed communication ... 46
6.10.Creation of output files ... 47
6.11.Time measurements ... 49

7.Implementing communication using PVM ... 50

7.1.init_par ... 50
7.2.make_topo ... 51
7.3.Exchange ... 52
7.4.Sum_all ... 54
7.5.Max_Nclds ... 56
7.6.Exit_par ... 58

8.Implementing communication using MPI ... 59

8.1.init_par ... 59
8.2.make_topo ... 60

SKIRON

8.3.Exchange 60
8.4.Sum_all 62
8.5.Max_Nclds 63
8.6.exit_par 65

9.The Parsytec CC-8 parallel computer........ 66

9.1.Architecture of the Parsytec-CC 66
9.1.1.Partitions 66
9.1.2.Virtual processors 68
9.2.The EPX operating system (Embedded Parix) 69
9.2.1.Capabilities 69
9.2.2.Message Passing 70
9.2.3.PowerPVM/EPX........ 71
9.2.4.EPX features 71
9.3.Parallel Implementation of the Eta code 72
9.3.1.Global Communication........ 72
9.3.2.Mixed Communication 76
9.4.Implementing communication using Parix 78
9.5.Numerical Results........ 81
Appendix 4-1 Execution instructions for the parallel Eta code on the Convex SPP 1600 platform with PVM and MPI 84
Appendix 4-2 Results of the parallel Eta code on the Convex SPP-600 platform with PVM and MPI 88
Appendix 4-3 Results of the parallel Eta code on the Convex SPP-2000 platform with MPI 95
Appendix 4-4 Results of the parallel Eta code on the Parsytec CC-8 platform with Parix and PVM 102
References........ 109

SKIRON

1.The SPP-1600 parallel computer

1.1.Introduction

The SPP1600 Convex parallel system belongs to the Exemplar systems class. Exemplar systems implement the massively parallel processing (MPP) by using scalable parallel processing (SPP) technology.

1.2.Architecture of the SPP-1600

An Exemplar system is a NUMA (Non Uniform Memory Access) architecture system, which can be thought of as a shared memory computer with two levels of memory latency. Memory available on the current hypernode constitutes the first level, and all other memory constitutes the second.

The SPP1600 system consists of a hypernode containing 8 HP PA-RISC 7100 processors. The hypernode contains four CPU blocks. Each CPU block consists of the following :

- Two PA-RISC processors.
- An associated data and instruction cache
- A CPU-private memory.
- Convex Toroidal Interface (CTI).

Each hypernode also contains one or more hypernode-private memories that can be accessed by any CPU within the hypernode, and one or more global memory blocks. The SPP1600 architecture uses two data caches. One 1st level 1-Mbyte on-chip and one 2nd level 2-Mbyte off-chip.

SKIRON

1.3.Subcomplexes

In the Exemplar systems the processes are executed in virtual machines which are called subcomplexes and are arbitrary choices of processors and shared memory. A subcomplex may consist of only one processor or of all the processors installed in the machine.

1.4.SPP Features

There are many features that were taken into account while designing the SPP architecture. Some of them are:

- Ease of use.
- High performance in fixed and floating point computations.
- Scalability.
- Exploitation of all available resources for problem solving.
- Reliability.

SKIRON

2.Parallel programming models

2.1.Introduction

During the last years two important evolutions took place in the scientific computing area. The first one concerns the development of massively parallel systems that can provide the computing power necessary for the solution of large problems. However, as the development of software did not keep up with the innovations in hardware, the need for the creation of new programming languages, new methods of design and new algorithms arose.

The second evolution relevant to the solution of scientific computing is the distributed processing, since the demands for large amount of calculations lead to the abandonment of the sequential machine and towards the distributed computation units linked together through a high speed network.

Distributed computing offers many advantages, such as:

- Low cost due to the utilization of already existing hardware.
- Improved performance.
- Fault tolerance

For the full exploitation of heterogeneous computer systems, various software packages have been developed. Among them are Express, P4, Linda, PVM, MPI, whose differences lie in the programming model used for their design, and in their performance.

SKIRON

2.2.Message Passing

In the process of message passing, the following two cases appear:

MPMD (Multiple Program - Multiple Data), where a set of computational worker processes perform work for one or more manager process. This approach is used when synchronization is required between worker processes.

SPMD (Single Program - Multiple Data), where the program spawns several identical processes that perform the same work independently on different data sets. In this approach, synchronization is often required between processes.

In general, message passing offers the opportunity of creating portable parallel programs. An application that uses message passing consists of several concurrent tasks each with its own data, using messages to communicate with one another. The structure of the parallel program and the method of data distribution are explicitly handled by the programmer, while the communication and the synchronization of the processes are achieved through message passing.

In the Exemplar systems, the programs that make use of message passing achieve the minimum overhead in process synchronization and data distribution due to the existence of low-latency interconnects.

Additionally, Exemplar systems are especially suited to the SPMD approach – which is used for the parallelisation of the weather prediction code because of their fast shared-memory communication, which minimizes synchronization delays.

The SPP architecture supports message passing from one thread within a process to another within the same or a different process. Messages are sent using dedicated hardware and shared memory copying. The messaging mechanism includes:

SKIRON

Messaging memory buffers

Messages are constructed in send-message buffers and transferred to receive-message buffers. All message buffers are 64-bytes long. Each processor in the hypernode has 8 buffers for sending messages.

Constructing a message in a send-message buffer

The message is constructed within the processor cache using store instructions, then flushed to memory when the message is complete.

Sending a message

The send-message mechanism is implemented by preconditioning the CPU agent to intercept a flush data cache instruction, FDC (Flush data cache) and force a different operation to be performed. The address specified by the FDC instruction is used as the message destination address.

Receiving a message

A message destination address is associated with each processor. Each message destination address has a message-receive queue, as well as two hardware registers used to manage the reading and writing in the queue.

Receiving the message interrupt

Part of the process of message receiving is the generation of an interrupt in the processor owning the receive-message queue. The interrupt is issued after the incoming data has been stored in the buffer and the queue write register has been incremented.

SKIRON

3.The PVM (Parallel Virtual Machine) programming environment

3.1.History of PVM

The development of the PVM programming environment began in the summer of 1989 at the Oak Ridge National Laboratory, by Vaidy Sunderam and Al Geist. The original edition (PVM 1.0) was initially used in limited applications inside the laboratory, and it was only after 1991 (PVM 2.0) that PVM started to be widely used in the area of scientific applications.

PVM is a software system that permits the programs of the users to refer to a heterogeneous collection of linked computers, in the same way they would refer to a virtual parallel computer. In this way, the parallel applications enjoy hardware independence (portability), similar execution environment and full exploitation of the existing computing resources.

PVM includes a series of routines that handle the user applications through message passing procedures. The user writes his application as a collection of cooperating tasks which access the PVM resources through a library of standard interface routines, in order to be initiated, synchronized, or to exchange messages and terminate successfully in the contexts of the execution of a parallel program. More specifically, any task in existence at any point in the execution of a concurrent application may start or stop other tasks or add or delete computers from the virtual machine. Any process may communicate and/or synchronize with any other. All the applications of the users as well as the control procedures of PVM are supervised by a special process called daemon.

SKIRON

3.2.PVM/GSM (Parallel Virtual Machine for Globally Shared Memory)

The PVM/GSM message-passing library is a PVM implementation that is finely tuned for running on Exemplar systems. Although PVM/GSM runs only on Exemplar systems, it can work with other networked hosts-if those hosts are each running a PVM daemon that is compatible with Oak Ridge National Lab's version 3.3.10 of PVM.

3.3.Capabilities

The PVM programming model offers a variety of capabilities such as:

a) Process numbering

Every process task in a PVM program is represented by a distinct integer number different from all other task numbers.

b) Process handling

PVM offers the capability to the users processes to change from normal processes to PVM processes and then to return to their original state. There exist routines for the addition and deletion of hosts from the virtual machine, for the communication between PVM processes and for the collection of information concerning the configuration of the virtual machine and the active PVM processes.

c) Error handling

In the case that a host has a problem, PVM automatically excludes it from the virtual machine.

d) Dynamic Process Groups

A process may belong to one or more groups. Every group can change dynamically at any time during the execution of a parallel program.

e) Signaling

Two modes of communication are provided between PVM processes.

SKIRON

f) <u>Communication</u>

PVM presumes that message exchange can take place between any two processes, without any limitations concerning the message size.

PVM communication model has the capability for

- Asynchronous blocking send
- Asynchronous blocking receive
- non - blocking receive

ensuring that the order of the received messages will be the same as the one with which they were sent.

g) <u>Uniform execution environment</u>

Different data encoding schemes are supplied, in order to handle data communication between heterogeneous machines.

h) <u>Support of programming languages</u> (C, C++, Fortran 77)

3.4.PVM Applications

The applications which have been developed on PVM during the last years encompass:

- Material Science
- Global Climate Modeling
- Atmospheric, oceanic, and space studies
- Meteorological forecasting
- 3-D groundwater modeling
- Weather modeling
- Superconductivity, molecular dynamics
- Monte Carlo CFD application
- 2-D and 3-D seismic imaging
- 3-D underground flow fields
- Particle simulation
- Distributed AVS flow visualization

SKIRON

4.The MPI (Message Passing Interface) programming environment

4.1. History of MPI

MPI (Message Passing Interface) was developed with the aim of becoming a widely used standard for the message passing parallel programming. So, it ought to be practical, portable, efficient and easy to use.

The MPIF (Message Passing Interface Forum) was created in 1993, with the participation of more than 40 organizations. From January 1993 until June 1994, a set of standards of message passing libraries was defined. In June 1995 MPIF created a second manual of MPI (1.1 edition) in which it corrected the errors of the previous, in an effort which started in January 1995.

Immediately after the defining of the basic library subroutines of MPI, many implementations for these libraries were made available, from (super) computer manufacturing companies targeting specific models, as well as public domain editions, referring to a huge variety of different computers. The most well known public domain edition for MPI is MPICH that was developed by the Argonne National Laboratories.

SKIRON

4.2. MPICH/GSM (Message Passing Interface for Globally Shared Memory)

The Convex Mpich is a high performance implementation of the 1.1 edition of MPI that has been developed for use in Exemplar systems. It is based upon the ANL/MPICH (Argonne National Lab's Mpich) implementation of MPI and offers improved capacity for message passing in the Exemplar globally shared memory architectures (GSM).

As a programming platform for message passing, MPI assures one of the most important aims for an application: Portability. The former is proved by the fact that MPI has been implemented and is available for virtually all (super) computer types, as well as for clusters of workstations.

4.3. Capabilities

The MPI programming model offers a variety of capabilities such as:

-Point-to-point communication.
-Blocking-NonBlocking communication.
-Synchronous or Asynchronous communication.
-Buffered or NonBuffered communication.
-Collective Communication
-Barriers
-Broadcast
-Gather-Scatter.
-Global Reduction functions
-Group Communication
-Group handling
-Communicator handling
-Process topologies.
-Virtual topologies.
-Topology constructors.
-Environmental and error handling.
-Profiling Interface

SKIRON

5.The Eta model

The Eta model has been developed at the US National Center for Environment Prediction (NCEP) based on a prior "minimum physics" version of the code written and tested at the Federal Hydrometeorological Institute and Belgrade University and at the Geophysical Fluid Dynamics Laboratory of Princeton University. It is the current-generation mesoscale Numerical Weather Prediction model running in production at NCEP for forecasts longer than 12 hours (Henderson et al., 1995). For further references see (Kallos, 1997) and citations herein. The numerical and physical parameterization schemes of the Eta model have been described in Janjic (Janjic, 1979, 1984, 1990, 1994), (Janjic and Mesinger, 1984) and Mesinger (Mesinger, 1973, 1976, 1984, 1997) (Mesinger and Arakawa, 1976), (Mesinger et al., 1988). In short the Eta model uses

- the Eta vertical coordinate (Mesinger, 1984) which permits step-like representation of mountains and quasi-horizontal coordinate surfaces.
- an Arakawa *E* grid and
- the Janjic (Janjic, 1984) horizontal advection scheme which imposes a strict control on false energy cascade (e.g. (Janjic and Mesinger, 1984)).

Eta was originally developed and optimized for vector-based architectures such as the Cray C90. Recently, the code was transformed into the two dimensional version in order to be suitable for parallelization on MIMD machines. In this paper we present the parallel implementation of the Eta model for distributed memory processors when their network is a mesh. The computation and communication analysis for a rectangular domain decomposition reveals that the

SKIRON

computation domain of each processor must be a square in order to achieve linear speedup and constant efficiency. The Parallel Virtual Machine (PVM), the Message Passing Interface (MPI), and the Parallel Extension to UNIX (PARIX) message passing libraries were used for the required communication. Finally, we present our results on a Parsytec CC-8 machine, using PVM and PARIX, and on a Convex Exemplar SPP-1600 machine with 8 nodes, using PVM and MPI.

5.1. Parallelization of the Eta model

In this section, we describe how to implement the Eta model on a message passing parallel computer. The most common techniques to introduce parallelism in atmospheric models is domain decomposition. The basic idea is to decompose the original domain into subdomains and assign each subdomain to a different processor. To keep the computations consistent with the sequential code inter-processor communication is usually needed.

The Eta model is organized into two major packages: dynamics and physics. The first solves the basic model equations while the second describes the effect of the physical processes. For the numerical computation of the dynamics, explicit schemes are used as they are ideally suited for parallel computation since they do not require the numerical solution of linear systems. The main advantage of the explicit schemes is that they require only local communication in contrast to the implicit schemes where global communication is needed. The disadvantage is that they must satisfy time-step constraints like the CFL condition. Regarding the physics, the computation for each grid point requires information from an entire column (radiative process) and to avoid communication it is important to keep such data local. Moreover, short-wave radiation can create load imbalances when a diurnal cycle is simulated as it depends on the location of the sun and on the cloud distribution. Thus, all grid columns can be considered to be independent of each other, allowing for a simple data decomposition. The model also contains a *semi-Langrangian* advection scheme (Ritchie, 1995). However, the technique requires access to data from nearby grid columns, since it is necessary to compute

SKIRON

the trajectory of a parcel of air. Therefore, a degree of local communication is introduced.

5.1.1.Domain Decomposition

A spatial domain can be decomposed in a number of different ways. However, for simplicity of algorithm implementation, the domain is usually divided into subdomains with simple and regular geometries. Next, by considering a mesh network topology for our processors, each subdomain is allocated to a processor. This mapping is crucial because it affects the communication, the degree of parallelism and the load balance among the processors. The ideal situation is the minimization of the ratio communication to computation time and a uniform work load across the processors throughout the execution of the algorithm. Taking into consideration the fact that the computations are implicit in the vertical direction, only the horizontal domain is decomposed. Let us consider the decomposition where the $\sqrt{N} \times \sqrt{N}$ domain of grid points is partitioned into p/q horizontal strips and each strip is again partitioned into q rectangles of equal size (Fig. 5.1.1.1(a)). Each of these rectangles has N/p points (Fig 5.1.1.1(b)).

SKIRON

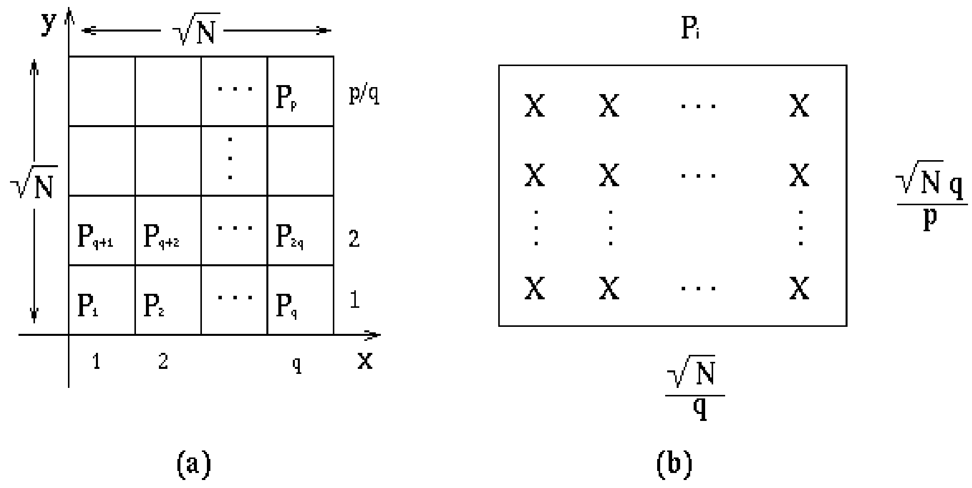


Figure 5.1.1.1: Domain decomposition and mapping to processors P_i, $1 \leq i \leq p$.

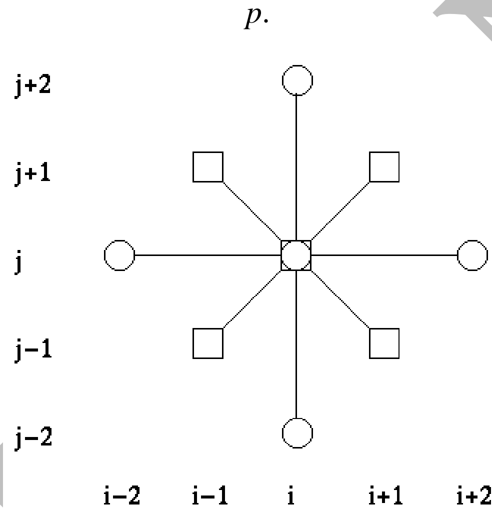


Figure 5.1.1.2: Stencil for the horizontal discretization

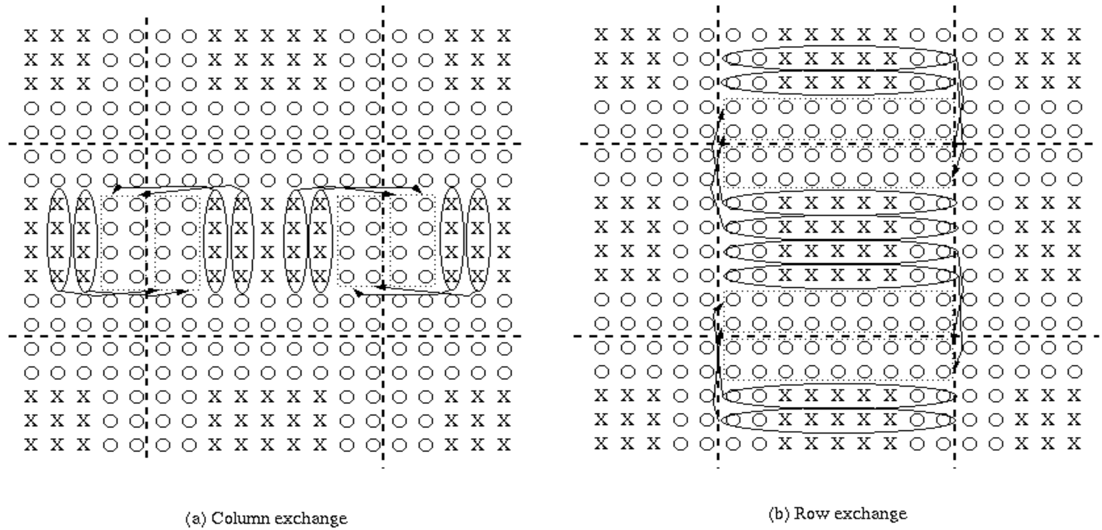


Figure 5.1.1.3: Stencil for the horizontal discretization

SKIRON

In every grid point the stencil shown in Fig. 5.1.1.2 for the horizontal discretization is applied (Ortega, 1988). Because of the structure of the computational stencil, processors have to communicate their boundary points to the four neighbouring processors after each scan of the domain. Therefore, each processor contains all the grid points corresponding to its domain, as well as those grid points which form its artificial boundary (in our case two rows/columns in each side). The communication pattern is shown in Fig. 5.1.1.3 and is carried out concurrently in all processors. To avoid communication between diagonal processors which contain only a common corner, the communication is carried out in two phases. In the first phase the column boundaries are exchanged with the left and right neighbour and in the second phase the row boundaries are exchanged with the upper and lower neighbour. Note, that the aforementioned phases may be performed in any order. The computational complexity, for each scan of the domain is analogous to the number of grid points in the rectangle (area) assigned to each processor P_i, namely

$$t_{comp}(P_i) = \frac{N}{P} t_c , \qquad (5.1.1.1)$$

where t_c is a machine dependent constant and is the constant amount of computation time per grid point. Note that $t_{comp}(P_i)$ is a decreasing function of the number of processors p. On the other hand, the communication time is the time required for each processor to communicate with its four neighbours. This time involves the exchange of data of two rows and two columns with the neighbour processors. The communication complexity, therefore, is analogous to the number of grid points which lie on the two rows/columns perimeter of the domain allocated to each processor, hence

$$t_{comm}(P_i) = 2\left[2t_s + \left(\frac{\sqrt{N}}{q} + \frac{q\sqrt{N}}{p}\right)t_w\right], \tag{5.1.1.2}$$

where t_s is the constant communication start-up time, t_w is a machine dependent constant and is the constant amount of the per word transfer time. From (5.1.1.1) and (5.1.1.2) it follows that we can determine the optimum value of q such that the ratio communication to computation is minimized. We have

$$r = \min_{1 \le q \le p} \frac{t_{comm}(P_i)}{t_{comp}(P_i)} = \min_{1 \le q \le p} \frac{2\left[2t_s + \left(\frac{\sqrt{N}}{q} + \frac{q\sqrt{N}}{p}\right)t_w\right]}{\frac{N}{p}t_c}$$

$$= \min_{1 \le q \le p} \frac{2\left[2t_s pq + t_w\left(p + q^2\right)\sqrt{N}\right]}{qNt_c}. \tag{5.1.1.3}$$

Letting $\frac{\partial r}{\partial q} = 0$ we find $\hat{q} = \sqrt{p}$ which is the optimum value for q. *Therefore, for the considered decomposition, it is best to assign to each processor a square block of grid side equal to* $\sqrt{\frac{N}{p}}$ (Boukas, Mimikou and Missirlis, 1997), (Kallos et al., 1997) . For the optimum value of q the corresponding minimum value of the ratio r is given by

$$r = \frac{4}{t_c}\left(t_s \frac{p}{N} + t_w \sqrt{\frac{p}{N}}\right). \tag{5.1.1.4}$$

SKIRON

From (5.1.1.4) we note the ratio r is composed of two terms and is governed by the first term as long as the ratio t_s/t_w is larger than or equal to the side of the square assigned to each processor, namely

$$\frac{t_s}{t_w} \geq \sqrt{\frac{N}{p}}. \qquad (5.1.1.5)$$

In fact, the first term of (5.1.1.4) depends linearly on p which means that for large number of processors, if (5.1.1.5) is satisfied, then r will behave as a linear function of p. On the other hand, if (5.1.1.5) does not hold, then the second term in (5.1.1.4) dominates in which case r is a very slowly increasing function of p. *In this case the ratio r is improved by an order of magnitude. In conclusion it is best to choose N and p such that*

$$\frac{t_s}{t_w} < \sqrt{\frac{N}{p}}. \qquad (5.1.1.6)$$

From the above analysis it is easily derived that the parallel run time needed for the computation of all the points in the domain is

$$T_p = t_c \frac{N}{p} + 4t_s + 4t_w \sqrt{\frac{N}{p}}. \qquad (5.1.1.7)$$

Under the assumption (5.1.1.6), the expressions for speedup and efficiency are as follows:

$$S_p = \frac{T_1}{T_p} > \frac{t_c N}{t_c \frac{N}{p} + 8t_w \sqrt{\frac{N}{p}}} = \frac{t_c p}{t_c + 8t_w \sqrt{\frac{p}{N}}}$$

and

$$E_p = \frac{S_p}{p} > \frac{t_c}{t_c + 8t_w \sqrt{\frac{p}{N}}}. \qquad (5.1.1.8)$$

From (5.1.1.8) we see that, for maintaining constant efficiency, N must be proportional to p. However, efficiency and speedup depend on the hardware constants t_c and t_w.

SKIRON

5.2. Parallel Implementation

In this section we present the principles, techniques and implementation details of the parallelization of the meteorological code Eta.

5.2.1. The parallelization principles and techniques

We followed four basic design principles:

1. Retain the original sequential code, as much as possible, for a number of reasons: the parallel code is recognisable by the sequential code developers; a smaller development effort is required as modifications are kept to a minimum; the parallel code performance may be directly compared with the sequential code.

We used the Single Program Multiple Data (SPMD) and the message-passing paradigms: the original data domain is partitioned into sub-domains, which are assigned to distinct processes, executing the same computations. The processes are spawned from the same executable, the computations of which are the same as the original sequential code, but parametrically restricted on data partitions. When communication between processes is required, message-passing operations are used. The results of individual processes are collected to form the output of the parallel application.

2. Minimize communication between processes for improving speed-up and efficiency.

Communication is needed when local computations require data in neighbouring partitions (halo points) and when computation of a value (e.g. maximum value in an array row) requires a distributed algorithm. For the former, we first determined the kernel and used overlapping of data partitions on halo points; we kept a local copy of halo points and only when required, we exchanged data by point-to-point bulk communication (complete rows or columns). For the latter, we redesigned the code, using known efficient distributed algorithms.

SKIRON

3. Develop portable parallel code for maintaining a single code for any execution environment. The execution environment depends on the particular message-passing environment (MPEs) used (e.g. PVM, MPI, Parix), as well as on the particular architecture (e.g. GC, CC, Exemplar).

PVM, MPI and to a certain degree Parix provide a virtual view of the underlying architectures, thus providing the necessary portability. We mainly had to deal with differences of MPEs related with process management and message passing operations. We developed high-level library procedures (e.g. Exchange row or column) which wrap low-level routines specific to each MPE. Porting the parallel code to a different MPE needs only relinking with the associated library.

4. Develop scalable code for the speed-up to be proportional to the number of available processors.

We concentrated our efforts on two factors on which scalability depends. The first was to keep the ratio of computations over communication constant. We specify regular equal orthogonal partitions parametrically. By allocating smaller data partitions to more processes, the computation load on each process decreases; also the exchange communication decreases as the number of halo points are fewer. Communication increases only when computing a value requiring a distributed algorithm. The second factor was the mapping of the virtual process topology on the architectures. As a data partition and the point-to-point process communication, required for exchanging data, determines a virtual grid process topology, we split data in as many partitions as processors available and map the process topology onto the processors. When the physical processor topology is important (e.g. in GC) we map logically neighbouring processes on physically neighbouring processors minimising latency.

In the sequel we elaborate on implementation details.

SKIRON

5.2.2. Determination of kernel and Halo points

Due to the *E-grid* morphology we can not consider the array columns as unified entities, but it is obligatory to view them as indivisible pairs of consecutive columns. The same stands for the case of array rows. In an effort to maintain the original philosophy of the model, the aforementioned constraint was satisfied in the parallel version. For this reason, it is necessary not only to inforce a pair-wise splitting of columns and rows, but also to ensure that the column kernel (row kernel) of each processor will begin with an odd column number (odd row number), and this first kernel column (first kernel row) will correspond to an odd numbered column (row) of the original array.

SKIRON

A second constraint of the model is that the number of columns *JM* and the number of rows *IM* of the mesh must be odd numbers. In this case, as well, the parallel code follows the imposed condition by setting the number of columns *JML* and the number of rows *IML* at each subdomain to be odd numbers.

Next, if we let $p \times q$ be a two-dimensional mesh of processors, and

$$k_j = \left[\frac{JM}{2}\right] \textit{ number of column pairs}$$

$$k_i = \left[\frac{IM}{2}\right] \textit{ number of row pairs,} \tag{5.2.2.9}$$

then it is easy to see that

$$JM = 2 * k_j + 1 \textit{ and } IM = 2 * k_i + 1. \tag{5.2.2.10}$$

Thus, each processor will hold

$$\left[\frac{k_j}{p}\right] \textit{column pairs and} \left[\frac{k_i}{q}\right] \textit{row pairs.} \tag{5.2.2.11}$$

However,

$$k_j = \left[\frac{k_j}{p}\right] \times p + r_j, \qquad 0 \le r_j \le p-1$$

and

$$k_i = \left[\frac{k_i}{q}\right] \times q + r_i, \qquad 0 \le r_i \le q-1 \tag{5.2.2.12}$$

That is, there is a surplus of r_j column pairs and r_i row pairs, which will eventually be uniformly allocated to the processors. Hence, each processor will hold

$$JML^* = 2 \times \left(\left[\frac{k_j}{p}\right] + \varepsilon_j\right), \quad \varepsilon_j \in \{0,1\} \tag{5.2.2.13}$$

consecutive columns and

SKIRON

$$IML^{*} = 2 \times \left(\left[\frac{k_i}{q} \right] + \varepsilon_i \right), \qquad \varepsilon_i \in \{0,1\} \tag{5.2.2.14}$$

consecutive rows of the original array.

So far, IML^* and JML^* represent kernel rows and kernel columns, respectively. However, each processor is surcharged with halo points. Thus, the grid points (kernel points and halo points) become

$$\begin{aligned} JML &= JML^* + halo_{left} + halo_{right} \\ IML &= IML^* + halo_{down} + halo_{up}, \end{aligned} \tag{5.2.2.15}$$

where $halo_{left}$, $halo_{right}$ is the number of left and right columns, respectively, consisting of halo points, whereas $halo_{down}$, $halo_{up}$ is the number of down and upper rows, respectively, consisting of halo points. According to our requirements $halo_{down}$, $halo_{up}$, $halo_{left}$, $halo_{right}$ must satisfy the following two conditions :

1. $halo_{down}$ and $halo_{left}$ must be even numbers, so that the first kernel row and the first kernel column to become odd numbered.
2. Due to the fact that the number of kernel columns and the number of kernel rows are even numbers (pair-wise splitting), $halo_{down}$ and $halo_{left}$ are also even numbers (due to the first constraint). To satisfy the second constraint (odd *IML*, odd *JML*) it is necessary to add an odd number. Therefore, $halo_{up}$ and $halo_{right}$ must be odd numbers in (5.2.2.15).

So far we have ignored the last column *JM* and the last row *IM* that can not form pairs due to the non existence of the *JM+1* column and the *IM+1* row, respectively, which is caused by the fact that *JM* and *IM* are defined to be odd numbers. These additional elements are surcharged to those processors that hold the *JM-1* column and/or the *IM-1* row. Such an addition, though, would result to the transformation of *JML* and *IML* to even numbers for these processors, which contradicts our constraint. For this reason we define for these processors

SKIRON

$$\text{halo}_{up} = \text{halo}_{up}-1$$

$$\text{halo}_{right} = \text{halo}_{right} - 1 \qquad (5.2.2.16)$$

so that *JML*, *IML* are odd numbers and are calculated from the same equation (5.2.2.15) in this case as well. Now that we have defined the constraints imposed on the halo points, we must combine the theoretical limitations with the actual implementation demands, i.e. we must find the real number of halo-columns and halo-rows that must be assigned to each processor. This is achieved with the following procedure.

For each array B that is used in the model, we search for expressions of the form

$$B(i\text{-}i_minus, \ldots), \quad B(i\text{+}i_plus, \ldots)$$

$$B(\ldots, j\text{-}j_minus), \quad B(\ldots, j\text{+}j_plus). \qquad (5.2.2.17)$$

Let

$$\begin{aligned} i_down &= \max(i_minus) \\ i_up &= \max(i_plus) \\ j_left &= \max(j_minus) \\ j_right &= \max(j_plus) \end{aligned}$$

then, we have

$$\begin{aligned} \text{halo}_{down} &= i_down + \text{mod}(i_down, 2) \\ \text{halo}_{left} &= j_left + \text{mod}(j_left, 2) \end{aligned} \qquad (5.2.2.18)$$

while

$$\begin{aligned} \text{halo}_{up} &= i_up + \text{mod}(i_up+1, 2) \\ \text{halo}_{right} &= j_right + \text{mod}(j_right+1, 2) \end{aligned} \qquad (5.2.2.19)$$

For the case of the parallel Eta model, the existence of additional limitations caused mainly by the Horizontal Advection Scheme's decomposition, leaded to a

SKIRON

number of up, down, left and right halo-points of, 2 rows,3 rows,4 columns, and 3 columns, respectively.

5.2.3 Communication

The execution of the model begins with the creation of the two-dimensional mesh topology for which the message passing procedures will take place. In the sequel, each processor will communicate with its top, down, left and right adjacent processor, named “upper”, “down”, “left” and “right”, respectively. Additionally, because of the need to identify each processor, the processors are numbered from *0* to *np-1*, and are related to the coordinates of the position they hold on the two-dimensional mesh topology.

For the purposes of message passing, a subroutine **Exchange** handles all the steps involved in the communication. Each message is formed by internal elements of the sending processor, then it is packed in a vector array, and is finally transmitted to the receiving processor, which will unpack the message and use it to update its halo points. According to which processor the messages will be sent to, and considering the fact that there is an overlapping of message so as to avoid communication with the diagonal processors, we have the following four variants of the Send subroutine:

Send_up, Send_down, Send_left, Send_right

Similarly, according to the processor from which the message has been sent, we have the following variants of the Receive subroutine:

Recv_down, Recv_up, Recv_right, Recv_left

Considering the fact that the proposed communication scheme consists of four Send calls, that can be performed in pairs concurrently, and four blocking Receive calls, the body of subroutine Exchange is either: call Send_up, call Send_down, call Recv_down, call Recv_up, call Send_left, call Send_right, call Recv_right, call Recv_left or call Send_left, call Send_right, call Recv_right, call Recv_left, call Send_up, call Send_down, call Recv_down, call Recv_up with additional two alternative forms.

SKIRON

Communication, expressed in calls of subroutine **Exchange**, is chosen to be performed in the beginning of the subprogram that requires it, in order to avoid repeated and unnecessary updating of the same halo points within loops. There are only a few cases in the model where message passing is applied in an inner code position, and are due to dependencies concerning temporary arrays only.

Moreover, communication is restricted to 2d and 3d arrays and is handled differently by each processor, in the sense that the amount of halo points exchanged by each processor depends on its position in the 2d-mesh, because processors laying on the perimeter of the mesh have fewer "valid" halo points than the internal ones. With the term "valid", we refer to those halo points that are essential for the execution, in contrast to the "dummy" halo points assigned to the processors from the side that they lack a neighbour, for reasons of consistency as well as security regarding pointer referencing.

Due to the previous considerations, calculations on an internal processor can include or exclude halo points, while on an external processor are restricted on kernel points only, for each side where no adjacent processor exists, because, an updating of the halo points in that case would just be a non required computational overload. On the contrary, the external processors can choose to update halo points for all sides that a neighbour exists.

In addition, communication is not always necessary in order to update correctly the halo points of a processor. Sometimes the nature of the local arrays helps avoiding a time-consuming message exchange. For this reason, in order to decide if communication is actually needed in a certain part of a program, we must first study the characteristics of the arrays that are declared. For the parallel Eta model, we have searched and categorized the local arrays as follows:

A local array is an

- **Input array** of a subprogram when it appears at least once in the right part of an expression without the existence of an updating occurrence in a previous statement of the same subprogram.

SKIRON

- **Output array** of a subprogram, when its values have been altered within the same subprogram.
- **Constant array** when its values do not change in any subprogram.
- **Updated array** when its halo points have the correct values, due to previous communication procedure or computations done on all the grid points.
- **Non-dependent** when all its elements (kernel and halo points) can be computed correctly by the same processor only with computations.

However, the halo points are only used to aid the computation of the kernel points, and do not have to be always updated, unless doing so will help to avoid communication. So, we have the following rules for applying interprocessor communication.

CASE 1. A whole subprogram can be executed strictly on kernel points and totally avoid communication if and only if

(a) all its input arrays are either Constant or Updated.

(b) the values of the halo-points are not needed to estimate the kernel points at any part of the subprogram.

CASE 2. A subprogram is forced to communicate if and only if at least one of its input arrays is not Updated and its halo points are needed for the computation of other arrays's kernel points.

CASE 3. A part of a subprogram can be executed only on kernel points of a processor if and only if the halo points of the Output arrays belonging to this part are not needed in a latter part of the same subprogram.

CASE 4. A part of a subprogram (or possibly the whole subprogram) can be executed for all the grid points (halo points and kernel points) of a processor if and only if it is used to create updated output arrays in order to help the next part or subprogram that will use them as input to avoid communication.

For a better understanding of our previous considerations, we present the following characteristic examples:

SKIRON

Example 1

Let us consider a sequential subroutine whose body is only the following for reasons of simplicity.

```
    DO 100 J=3, JM-2
        DO 100 I=1, IM
            A(I,J) = B(I,J) * c + A(I,J)
    100 CONTINUE
```

The estimation of element *A(I,J)* depends only on the corresponding values of *B* and *A* itself. Therefore, because of CASE 1(b) no communication is needed, and the calculations can be restricted to the kernel points. Consequently, the equivalent parallel form will be:

```
    DO 100 J=j_kern_start_3, j_kern_end_JM_2
        DO 100 I=i_kern_start_1, i_kern_end_IM
            A(I,J) = B(I,J) * c + A(I,J)
    100 CONTINUE
```

where

j_kern_start_3 is the third kernel column of the processor if it does not have a left neighbour otherwise is the first kernel column

j_kern_end_JM_2 is the third from the end kernel column of the processor if it doesn't have a right neighbour otherwise is last kernel column

i_kern_start_1 is the first kernel element of the processor's *j'th* column

i_kern_end_IM is the last kernel element of the processor's *j'th* column.

Example 2

Let us consider the following sequential part of code

SKIRON

```
DO 110 J=1, JM
    DO 110 I=1, IM
        A(I,J) = AETA(I) * B(I,J)
110 CONTINUE

DO 120 J=3, JM-2
    DO 120 I=2, IM-1
        C(I,J) = D(I,J) + B(I,J+1)/(A(I+IVW(J),J)+A(I+IVE(J),J))
120 CONTINUE
```

where
AETA is a vector with known elements, *B* is an Input array, *IVE(J)*=J modulo 2 and *IVW(J)*=IVE(J) - 1.

Let us suppose that *B* is not an Updated array. Then, we notice from loop "Do 120", that the calculation of *C(i,j)* for some *j*, may depend on the values of *A(i-1, j)*, *A(i+1, j)*, and *B(i, j+1)*. These *A*, *B* elements, though, do not always belong to the processor's kernel. This means, that in the calculations of loop "DO 120" we need the correct values of each processor's halo points as well as the kernel points. The values of *A* could be calculated in loop "Do 110" for the whole grid. However, the value of *A* at each halo point depends on the corresponding value of *B*, which is not previously computed, i.e. *A* is a Dependent array. It is obvious, that the situation is the one described in CASE 2. Hence, we must get the correct values of *B* via the message passing procedure. In doing so *B* will be converted into an Updated array, and *A* into a Non-dependent array. Thus, after the communication, CASE 2 leads to CASE 4 for loop "DO 110" and to CASE 3 for loop "DO 120". The parallel form is then the following:

```
CALL Exchange_2d(B)
DO 110 J=j_startg_1, j_endg_JM
    DO 110 I=i_startg_1, i_endg_IM
        A(I,J) = AETA * B(I,J)
```

SKIRON

```
110 CONTINUE
DO 120 J=j_kern_start_3, j_kern_end_JM_2
   DO 120 I=i_kern_start_2, i_kern_end_IM_1
      C(I,J) = D(I,J) + B(I,J+1)/(A(I+IVW(J),J)+A(I+IVE(J),J))
120 CONTINUE
```

where

Exchange_2d is the communication handler Exchange marked 2d because B is a two dimensional array.

j_startg_1 is the first kernel column of the processor if it doesn't have a left neighbour otherwise is the first grid column

j_endg_JM is the last kernel column of the processor if it doesn't have a right neighbour otherwise is the last grid column

i_startg_1 is the first kernel element of the processor's *j'th* column if it doesn't have a down neighbour otherwise is the first grid element

i_endg_IM is the last kernel element of the processor's *j'th* column if it doesn't have an upper neighbour otherwise is the last grid element

5.2.4. Output Data Acquisition

During the execution of the sequential program, a special subroutine creates, in predefined time periods, the output files of the meteorological model. Executing the same subroutine in parallel results in acquiring a number of output files analogous to the number of processors that participated. These files contain subsets of what would normally be the output of the model. A set of programs, using steps quite similar to the ones taken for data decomposition, act as a postprocessing part, joining the output data of parallel Eta in files, such as to produce the appropriate output.

5.3.Numerical Results

The aforementioned data partitioning and message passing techniques have been successfully tested and validated in the shared memory platform Exemplar SPP-1600. The programming models that were selected for the implementation are

SKIRON

PVM and MPI because they combine machine independence, portability and compatibility.

Figures 5.3.4-5.3.7 represent our numerical results of the model as they were estimated for a 48h run of 1920 steps on a $121 \times 161 \times 32$ grid, with *4*, *6* and *8* processors, respectively. Times are measured in seconds in all cases.

The reduction in time of the parallel code as compared to its sequential version is significant (Fig. 5.3.4). The speedup is almost linear for all cases (Fig. 5.3.5) except the PVM version, where there is a discrepancy for $p>6$. The efficiency of all cases with exception again of the PVM version is above *0.8* (Fig. 5.3.5). Fig. 5.3.6 and 5.3.7 show the behaviour of the computation and communication time. The reduction in communication time of MPI as compared to PVM is significant.

SKIRON

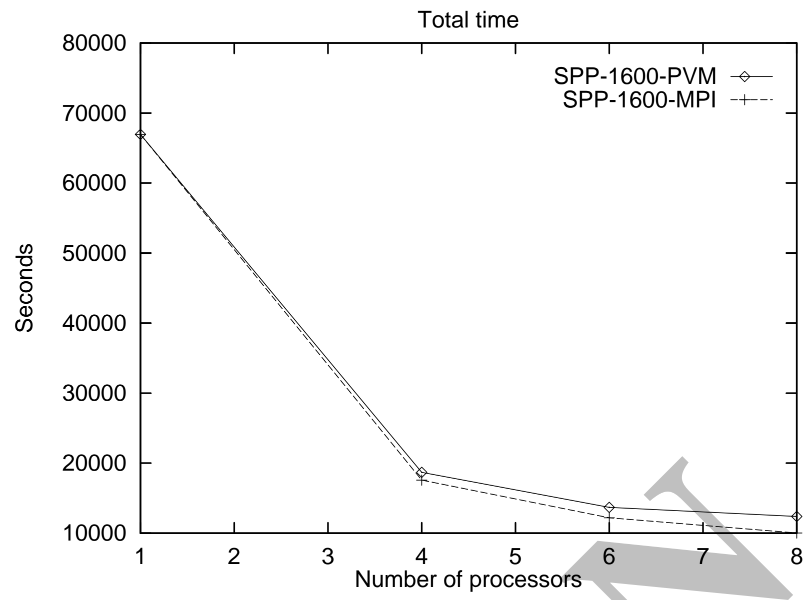


Figure 5.3.4: Parallel run time

SKIRON

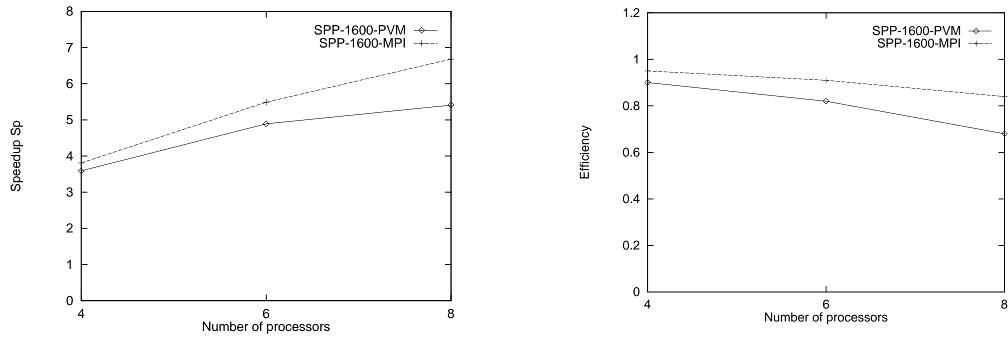


Figure 5.3.5: Speedup-Efficiency

SKIRON

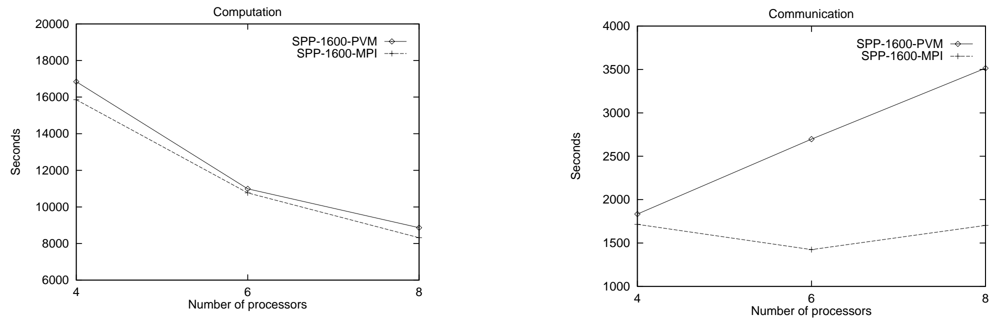


Figure 5.3.6: Computation-Communication

SKIRON

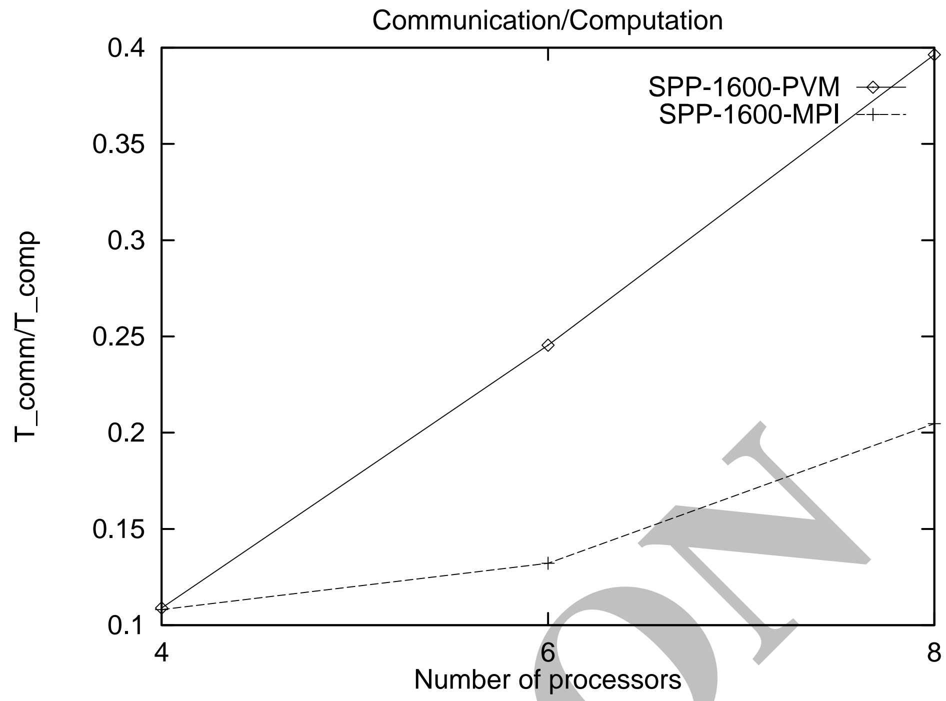


Figure 5.3.7: Communication/Computation

SKIRON

6.Introduction to the parallel code

In this section we present the programs and subroutines that were used to parallelize the meteorological code ETA for the SPP-1600 Convex machine, using the PVM and the MPI programming environment.

6.1. Creation of the input files

Name subsplit.f

Description

The first step of a parallel program is to allocate the data to the available processors. For the parallel meteorological code Eta this demand is satisfied by using as a preprocessing part, a program called subsplit, that handles the data mapping to the processors (not taking into account the input data relative to the boundary conditions of the model).In particular, program subsplit reads the input files of the sequential model and distributes their contents to smaller processor dependent files that in a latter step will be used as input to the parallel code. Consequently, a partitioning of data via program subsplit occurs each time is needed to change the set of input data or the number of the participating processors.

Call

Program execution is accomplished with the command:

```
Subsplit
```

and must precede the execution of the parallel program.

SKIRON

Parameters

In order to modify the dimensions of the processor mesh, parameters N_X_GRID and N_Y_GRID must be updated with the new values in file parallel.h. In particular, subsplit.f includes the following files:

split.h : defines parameters and variables relevant to the input data.

local.h : defines temporary variables used for the data mapping.

parallel.h : defines the relevant to the parallel program declarations, allowing to
modify the following values:

N_X_GRID: number of processors in the horizontal axis
N_Y_GRID: number of processors in the vertical axis
NUM_PROCS = N_X_GRID x N_Y_GRID (number of processors)

6.2. Creation of the input files relative to the boundary conditions

Name split_boco.f

Description

Program split_boco.f handles the data mapping to the processors, for those input data files that correspond to the boundary conditions imposed on the execution of the model. These data are stored in files created in predefined time periods from the preprocessing part. Split_boco.f produces on arrival the files that will be read in a latter step from the subroutine bocoh of the parallel program, without demanding the completion of its execution prior to the execution of the parallel program. In this way, the data are allocated to the processors as soon as the first file is made availiable and the parallel program can start its execution. If a new input file is created, then program split_boco.f will act independendly to the execution status.

SKIRON

Also, in case that the data mapping is not completed at a certain timestep, whereas the parallel program has reached the point of reading the new boundary data, then it will wait until split_boco.f terminates.

Call

Program execution is accomplished with the command:

Sboco k

where k=1(1)TEND/TBOCO.

Parameters

Program split_boco.f includes the following files:

split.h : defines parameters and variables relevant to the input data.
local.h : defines temporary variables used for the data mapping.
parallel.h : defines the relevant to the parallel program declarations, allowing to
modify the following values:

N_X_GRID: number of processors in the horizontal axis
N_Y_GRID: number of processors in the vertical axis
NUM_PROCS = N_X_GRID x N_Y_GRID (number of processors)

SKIRON

6.3. Storage of significant values

Name save_values

Description

Subroutine save_values is called in order to store in a temporary file (save.dat) all the parameters which will be needed in the post-processing part following the execution of the parallel program, where the output files will be merged together according to a proposed data composition technique (program submerge.f).

Call

Subroutine save_values is called at the end of subroutine INIT, with the command:

call save_values

Parameters

The parameters whose values are stored in a temporary file from the save_values routine are the following:

IDAT, IHRST, PT, ETA, DETA, ALSL, SPL, AETA, DFL, NTSD, NTSTM, NHOUT, DY, TSPH

SKIRON

6.4. Initialization of the parallel program

Name init_par

Description

During the execution of a parallel program each processor is assigned a set of data that differ from the ones of the other processors. Thus, it is needed to distinguish one processor from another. This is achieved by numbering the processors from *0* to *p-1* and relating them to the coordinates (my_x, my_y) of the position they hold on the two-dimensional mesh topology.

Call

Subroutine init_par is called from the main program, before subroutine INIT, with the command:

call init_par

Parameters

my_id	: the processor's number
my_x	: the processor's position according to the horizontal axis
my_y	: the processor's position according to the vertical axis.
dim_x	: the processor mesh dimension according to the horizontal axis.
dim_y	: the processor mesh dimension according to the vertical axis.
np	: the number of processors.

SKIRON

6.5.Creation of the two dimensional mesh topology

Name make_topo

Description

Subroutine make_topo creates the two dimensional mesh topology through which the message passing procedures will take place. Each processor will communicate with his top, down, left and right adjacent processor, named 'upper', 'down', left and 'right', respectively.

Call

Subroutine make_topo is called by the main program with the command:

```
call make_topo
```

Parameters

The following parameters represent the four neighbouring processors of a processor on the two dimensional mesh.

Upper : Upper neighbour
Down : Lower neighbour
Left: Left neighbour
Right : Right neighbour

SKIRON

6.6.Termination of the parallel program

Name exit_par

Description

Subroutine exit_par accomplishes the normal termination of the parallel program by freeing the processors participating in the execution.

Call

It is called at the end of the main program with the command

```
call exit_par
```

Parameters

The parameters used vary according to the parallel implementation environment.

6.7.Local communication

Name exchange

Description

Subroutine exchange handles all the steps involved in the local communication. In particular, according to the selected communication mode (synchronous/asynchronous communication), subroutine exchange consists of a sequence of Send and Receive calls executed by any processor that succeeds on satisfying certain conditions.

SKIRON

Local communication is applied when there is a need to update the halo points of one or more arrays owned by a processor. Normally, this would lead to a communication pattern involving the eight neighbouring processors of each processor on the mesh topology. In order to avoid communication between diagonal processors that contain only a common corner, communication is carried out in two phases.

- First Phase: The halo-columns are exchanged with the left and right neighbour.
- Second Phase: The halo-rows are exchanged with the upper and lower neighbour.

The proposed communication scheme consists of four asynchronous Send calls, that can be performed in pairs concurrently, and four blocking Receive calls. Due to the fact that the two phases can be performed in any order, there are two possible ways of arranging the Send and Receive calls. The order that has been selected for the implementation of the local communication scheme is the following:

```
    { Exchange }

if   has upper   Send (up)
if   has down    Send (down)

if   has down    Receive (down)
if   has upper   Receive (up)

if   has left    Send (left)
if   has right   Send (right)

if   has right   Receive (right)
if   has left    Receive (left)
```

SKIRON

Each one of the Send routines undertakes to store the message in a vector array (packing) and then send it through the mesh topology. Similarly, each one of the Receive routines receives the message and uses its values (unpacking) in order to update the halo points of the array that caused the genesis of the local communication at that point.

Call

Due to the fact that all the arrays involved in local communication procedures have either two or three dimensions, subroutine exchange can take on of the following forms :

- Exchange_2d, if the array participating in the communication has two dimensions.
- Exchange_3d, if the array participating in the communication has three dimensions.

Thus, local communication is started with the commands:

a) call Exchange_2d (AR2D), when AR2D is the two dimensional array AR2D(I,J) elements of which will be exchanged.

b) call Exchange_3d (AR3D, LL), when AR3D is the three dimensional array AR3D(I,J,L) elements of which will be exchanged for L=1(1)LL.

Parameters

The parameters used to set the number of halo points that will be exchanged with each one of the upper, lower, left and right processors, are respectively

i_BD_U : Number of upper halo-rows

SKIRON

j_BD_D : Number of lower halo-rows

j_BD_L : Number of left halo-columns

j_BD_R : Number of right halo-columns

6.8. Global Communication

Name Sum_all

Description

The introduction of parallelism on subroutine HZADV2 revealed a communicational need between all the processors on the mesh topology (global communication). In particular, each call of subroutine HZADV2 involved a computation of four global sums, that could not be performed independantly at each processor, as they demanded the knowledge of points that did not belong to the same or neighbouring processors.

The subprogram which was developed, namely SUM_all, conducts the gathering of all the partial sums from the processors the computation and the scattering of all the computed global sums in one step in order to avoid surcharging the communication time with the startup time for four separate messages. In particular, assuming that the number of processors is *n* and the processor which will collect the partial sums is processor *p*, then the algorithm of the subprogram that implements global communication is the following:

{ SUM-all }

if I am processor p then

for all processors $q \neq p$ *do*

Receive (q, Psum1, Psum2, Psum3, Psum4)

SKIRON

$$Sum1 = \sum_n Psum1$$

$$Sum2 = \sum_n Psum2$$

$$Sum3 = \sum_n Psum3$$

$$Sum4 = \sum_n Psum4$$

Send (q, Sum1, Sum2, Sum3, Sum4)

else

Send (p, Psum1, Psum2, Psum3, Psum4)

Receive (p, Sum1, Sum2, Sum3, Sum4)

Call

call Sum_all (x, y, s, r)

where x, y, s, r are the partial sums.

Parameters

The messages which are exchanged consist of groups of four partial sums, and are addressed according to the parameters:

my_x : the position of the processor on the horizontal axis.
my_y : the position of the processor on the vertical axis.
dim_x : the dimension of the mesh topology according to the horizontal axis.
dim_y : the dimension of the mesh topology according to the vertical axis.

6.9. Mixed communication

Name Max_Nclds

SKIRON

Description

In subroutines FST88 and SWR93 of the model, there was a need to know the maximum value of vector NCLDS. The values of this vector, though, could not be computed localy at each processor for the whole vector, as they dependend on values belonging to different processors. For this reason, a new subrprogram was implemented. This subprogram is named Max_Nclds and undertakes to compute the global maximum of vector NCLDS requesting as input the local maximum values from all the processors that belong to the same column of the two dimensional mesh. calculations.

Let $x \times y$ be a mesh of processors and p_i the processor that will gather the local maximums of the *i-th* column $(1 \leq i \leq x)$. Then, subprogram Max_Nclds can be expressed in pseudocode as follows

{ Ìax_Nclds }

if I am processor p_i *then*

for all processors $q_i \neq p_i$ *that belong in the ith column of the mesh*

do

Receive $(q_i, PMax_i)$

$Max_i = Max(PMax_i)$

Send (q_i, Max_i)

else

Send $(p_i, PMax_i)$

Receive (p_i, Max_i)

Call

call Max_Nclds (K)

SKIRON

where K is the local maximum of each processor.

Parameters

my_x **:** the position of the processor on the horizontal axis.

my_y **:** the position of the processor on the vertical axis.

dim_x **:** the dimension of the mesh according to the horizontal axis.

dim_y **:** the dimension of the mesh according to the vertical axis.

6.10.Creation of output files

Name submerge.f

Description

During the execution of the sequential program, subroutine OUTVAR creates in predefined time periods the output files of the model. Executing the same subroutine in parallel results in acquiring a number of output files analogous to the number of processors that participated. These files contain subsets of what would normally be the output of the model. Program submerge, acting as a postprocessing part, joins the output data of the parallel program in files, so as to produce the appropriate output.

SKIRON

Call

Program submerge is executed with the command

Submerge k

where k=1(1) TEND/TBOCO

and its execution does not depend on the termination of the parallel program, as far as there exists at least one set of output files to join.

Parameters

Program submerge.f includes the following file:

parallel.h :contains all the statements relevant to the parallel program, with the capacity of changing values:
N_X_GRID : number of processors in the horizontal axis
N_Y_GRID : number of processors in the vertical axis
NUM_PROCS : N_X_GRID x N_Y_GRID (number of processors)

6.11. Time measurements

Name ekpa_time

Description

The lack of uniform timing schemes on the platforms used for the implementation lead to the creation of a intermediate timing function whose content will vary according to the platform in use, but whose call will always use the same name and syntax.

Function Ekpa_time returns in all cases the wall-clock time taken to execute the model in seconds.

Call

SKIRON

Function Ekpa_time is called every time there is a timing issue with the command

now = ekpa_time()

where now: A double precision floating point number (REAL*8).

SKIRON

7.Implementing communication using PVM

7.1. init_par

Name init_par

Description

Spawning of processes and enrolment in the group “topo”.

Routines

a) pvmfmytid: Returns the tid of the calling process

call pvmfmytid (tid)

tid : 32-bit positive integer task identifier number unique for each process

b) pvmfjoingroup: Enrolls the calling process in a named group

call pvmfjoingroup (group, inum)

group : character string group name of an existing group
inum : integer instance number returned, different for each procedure

c) pvmfspawn : Starts new pvm processes

call pvmfspawn (task, flag, where, ntask, tids, numt)

task : character string containing the executable file name

SKIRON

flag : integer specifying spawn options

where : host or architecture where the processes will be started

ntask : the number of copies of the executable to start up

tids : integer array which, on return will contain the tids of the new processes

numt : integer returning the actual number of tasks started

d) pvmfbarrier: Blocks the calling process until all processes in a group have called it

call pvmfbarrier (group, count, info)

group :character string group name

count :integer number of group members that must call pvmfbarrier before they are all released

info : integer status code returned by the routine

7.2. make_topo

Name make_topo

Description

Retrieval, for each process, of the tids of the processes which belong to the group "topo" and are executed by neighbouring processors.

Routines

a) Pvmfgettid: Returns the tid of the process identified by a group name and instance number

call pvmfgettid (group, inum, tid)

SKIRON

group : character string containing the name of an existing group

inum :integer instance number of the process in the group

tid : integer task identifier returned

7.3. Exchange

Name Exchange

Description

Implements local communication.

Routines

a) pvmfinitsend: Clears default send buffer and specifies message encoding

call pvmfinitsend (encod, bufid)

encod : integer specifying the next message's encoding scheme

bufid : integer message buffer identifier

b) pvmfpack: Packs the active message buffer with arrays of prescribed data type

call pvmfpack(what, xp, nitem, stride, info)

what : integer specifying the type of data being packed

xp : pointer to the beginning of a block of bytes

nitem : the total number of items to be packed

stride : the stride to be used when packing the items

info : integer status code returned by the routine

c) pvmfsend: Sends the data in the active message buffer

SKIRON

call pvmfsend (tid, msgtag, info)

tid : integer task identifier of destination process
msgtag : integer message tag supplied by the user
info : integer status code returned by the routine

d) pvmfrecv : Receives a message

call pvmfrecv (tid, msgtag, info)

tid : integer task identifier of sending process supplied by the user
msgtag : integer message tag supplied by the user
info : integer returning the value of the new active receive buffer identifier

e) pvmfunpack: Unpacks the active message buffer into arrays of prescribed data type

call pvmfunpack (what, xp, nitem, stride, info)

what : integer specifying the type of data being unpacked
xp : pointer to the beginning of a block of bytes
nitem : the total number of items to be unpacked
stride : the stride that was used when packing the items
info : integer status code returned by the routine

f) pvmffreebuf: Disposes of a message buffer

call pvmffreebuf (bufid, info)

bufid : integer message buffer identifier
info : integer status code returned by the routine

SKIRON

7.4. Sum_all

Name: Sum_all

Description

The algorithm which is executed in order to conduct global communication on PVM, has been chosen to use asynchronous message send and blocking receive.

Routines

a) pvmfinitsend: Clears default send buffer and specifies message encoding

call pvmfinitsend (encod, bufid)

encod : integer specifying the next message's encoding scheme
bufid : integer message buffer identifier

b) pvmfpack: Packs the active message buffer with arrays of prescribed data type

call pvmfpack(what, xp, nitem, stride, info)

what : integer specifying the type of data being packed
xp : pointer to the beginning of a block of bytes
nitem : the total number of items to be packed
stride : the stride to be used when packing the items
info : integer status code returned by the routine

c) pvmfsend: Sends the data in the active message buffer

call pvmfsend (tid, msgtag, info)

tid : integer task identifier of destination process

SKIRON

msgtag : integer message tag supplied by the user

info : integer status code returned by the routine

d) pvmfrecv : Receives a message

call pvmfrecv (tid, msgtag, info)

tid : integer task identifier of sending process supplied by the user

msgtag : integer message tag supplied by the user

info : integer returning the value of the new active receive buffer identifier

e) pvmfunpack: Unpacks the active message buffer into arrays of prescribed data type

call pvmfunpack (what, xp, nitem, stride, info)

what : integer specifying the type of data being unpacked

xp : pointer to the beginning of a block of bytes

nitem : the total number of items to be unpacked

stride : the stride that was used when packing the items

info : integer status code returned by the routine

f) pvmffreebuf: Disposes of a message buffer

call pvmffreebuf (bufid, info)

bufid : integer message buffer identifier

info : integer status code returned by the routine

7.5. Max_Nclds

Name: Max_Nclds

SKIRON

Description

The algorithm which is executed in order to conduct mixed communication on PVM, has been chosen to use asynchronous message send and blocking receive.

Routines

a) pvmfinitsend: Clears default send buffer and specifies message encoding

call pvmfinitsend (encod, bufid)

encod : integer specifying the next message's encoding scheme
bufid : integer message buffer identifier

b) pvmfpack: Packs the active message buffer with arrays of prescribed data type

call pvmfpack(what, xp, nitem, stride, info)

what : integer specifying the type of data being packed
xp : pointer to the beginning of a block of bytes
nitem : the total number of items to be packed
stride : the stride to be used when packing the items
info : integer status code returned by the routine

c) pvmfsend: Sends the data in the active message buffer

call pvmfsend (tid, msgtag, info)

tid : integer task identifier of destination process
msgtag : integer message tag supplied by the user
info : integer status code returned by the routine

SKIRON

d) pvmfrecv : Receives a message

call pvmfrecv (tid, msgtag, info)

tid : integer task identifier of sending process supplied by the user
msgtag : integer message tag supplied by the user
info :integer returning the value of the new active receive buffer identifier

e) pvmfunpack: Unpacks the active message buffer into arrays of prescribed data type

call pvmfunpack (what, xp, nitem, stride, info)

what : integer specifying the type of data being unpacked
xp : pointer to the beginning of a block of bytes
nitem : the total number of items to be unpacked
stride : the stride that was used when packing the items
info : integer status code returned by the routine

f) pvmffreebuf: Disposes of a message buffer

call pvmffreebuf (bufid, info)

bufid : integer message buffer identifier
info : integer status code returned by the routine

7.6. Exit_par

Name exit_par

SKIRON

Description

Termination of the parallel program

Routines

a) pvmflvgroup: Unenrolls the calling process from a named group

call pvmflvgroup (group, info)

group : character string group name of an existing group
info : integer status code returned by the routine

b) pvmfexit: Informs the daemon that the calling process is leaving PVM

call pvmfexit (info)

info : integer status code returned by the routine

SKIRON

8.Implementing communication using MPI

8.1. init_par

Name init_par

Description

Preparation for program execution with MPI

Routines

a) MPI_init: MPI initialization

call MPI_Init (ierror)

ierror : error code

b)MPI_Comm_Rank: Initialisation of the process communication field.

call MPI_Comm_Rank (comm, rank, ierror)

comm : the communicator
rank : integer number returned, different for each process
ierror : error code

SKIRON

8.2. make_topo

Name make_topo

Description

In the make_topo routine MPI commands have not been used, as the creation of a virtual topology is not necessary for communication. The virtual topology is created through the logical variables has_upper, has_down, has_left, has_right.

8.3.Exchange

Name Exchange

Description

The algorithm which implements local communication in MPI, has been selected to use buffered send and blocking receive.

Routines

a) MPI_Buffer_Attach: Provides to MPI a buffer in the user's memory to be used for buffering outgoing messages.

call MPI_Buffer_Attach (buffer, size, ierror)

buffer : initial buffer address.
size : the size of the buffer in bytes
ierror : error code

b)MPI_Buffer_Detach: Detaches the currently associated with MPI buffer

call MPI_Buffer_Detach ((buffer, size, ierror)

SKIRON

buffer : initial buffer address.

size : the size of the buffer in bytes

ierror : error code

c) MPI_BSend : Send in buffered mode

call MPI_BSend (buf,count,dtype, dest, tag, comm, ierror)

buf : initial address of send buffer

count : number of elements in send buffer

dtype : datatype of each send buffer element

dest : rank of destination

tag : message tag

comm : the communicator

ierror : error code

d) MPI_Recv : Blocking receive

call MPI_Recv (buf, count, dtype, source, tag, comm, status, ierror)

buf : initial address of receive

count : number of elements in receive buffer

dtype : datatype of each receive buffer element

source : rank of source

tag : message tag

comm : the communicator

status : status object

ierror : error code

SKIRON

8.4. Sum_all

Name Sum_all

Description

The algorithm which implements global communication in MPI, has been selected to use buffered send and blocking receive.

Routines

a) MPI_Buffer_Attach: Provides to MPI a buffer in the user's memory to be used for buffering outgoing messages.

call MPI_Buffer_Attach (buffer, size, ierror)

buffer : initial buffer address.
size : the size of the buffer in bytes
ierror : error code

b)MPI_Buffer_Detach: Detaches the currently associated with MPI buffer

call MPI_Buffer_Detach ((buffer, size, ierror)

buffer : initial buffer address.
size : the size of the buffer in bytes
ierror : error code

c) MPI_BSend : Send in buffered mode

call MPI_BSend (buf,count,dtype, dest, tag, comm, ierror)

SKIRON

buf : initial address of send buffer
count : number of elements in send buffer
dtype : datatype of each send buffer element
dest : rank of destination
tag : message tag
comm : the communicator
ierror : error code

d) MPI_Recv : Blocking receive

call MPI_Recv (buf, count, dtype, source, tag, comm, status, ierror)

buf : initial address of receive
count : number of elements in receive buffer
dtype : datatype of each receive buffer element
source : rank of source
tag : message tag
comm : the communicator
status : status object
ierror : error code

8.5. Max_Nclds

Name Max_Nclds

Description

The algorithm which implements mixed communication in MPI, has been selected to use buffered send and blocking receive.

SKIRON

Routines

a) MPI_Buffer_Attach: Provides to MPI a buffer in the user's memory to be used for buffering outgoing messages.

call MPI_Buffer_Attach (buffer, size, ierror)

buffer : initial buffer address.
size : the size of the buffer in bytes
ierror : error code

b)MPI_Buffer_Detach: Detaches the currently associated with MPI buffer

call MPI_Buffer_Detach ((buffer, size, ierror)

buffer : initial buffer address.
size : the size of the buffer in bytes
ierror : error code

c) MPI_BSend : Send in buffered mode

call MPI_BSend (buf,count,dtype, dest, tag, comm, ierror)

buf : initial address of send buffer
count : number of elements in send buffer
dtype : datatype of each send buffer element
dest : rank of destination
tag : message tag
comm : the communicator

SKIRON

ierror : error code

d) MPI_Recv : Blocking receive

call MPI_Recv (buf, count, dtype, source, tag, comm, status, ierror)

buf : initial address of receive
count : number of elements in receive buffer
dtype : datatype of each receive buffer element
source : rank of source
tag : message tag
comm : the communicator
status : status object
ierror : error code

8.6. exit_par

Name exit_par

Description

Termination of the processes

Routines

a)MPI_Finalize: Termination of the use of MPI states by the process

call MPI_Finalize (ierror)

ierror : error code

SKIRON

9.The Parsytec CC-8 parallel computer

The Parsytec-CC parallel computing system belongs to the class of cognitive systems and has been designed to offer solutions in the area of cognitive automation for industrial and scientific applications.

9.1. Architecture of the Parsytec-CC

The system architecture follows the principles of MIMD computers. A CC system is usually composed of the following main parts which, depending on the way they are linked and arranged, form a CC system of variable size and topology.

***IO-MODULE*:** It is a processor node that consists of one or more processors with peripheral I/O.

***P- MODULE*:** Purely processing node with one or more than one processors, local disc and an HS-Link interface.

***ROUTER-MODULE*:** The nodes are connected by Router-Modules.

More in depth, the Parsytec-CC computing system consists of a group of interlinked processor nodes, each of which has its own local memory. Each node is a RISC PowerPC-604 processor, while the interconnection is achieved through a serial high speed communication link (HS-Link). In addition, high performance routers offering a deadlock - free mechanism, suitable for the establishment of a high performance communication network, are used.

9.1.1.Partitions

A partition is a group of processors allocated exclusively to one user. Usually, a partition is only a part of the whole system, although it can include all the machine. The partitions can overlap, meaning that a partition can constitute a subset of a

SKIRON

bigger partition. In that case, the allocation of the smaller partition to one user may lead to the bigger partition being not allocatable to another user.

Every processor node belonging to a partition functions, from the beginning till the end of the parallel application, independently of the others, possessing its own executable program copy in its local memory (Single Program Multiple Data). At the local memory of each processor node only that node has access. In this way, no other node except the specified can refer to the variables and the data that it uses.

Different program execution by different nodes is achieved

- by the execution of different code parts of the same program copy
- by the application of the same commands on different data

The identity and the position of each node inside the partition is necessary so that each node will know what exactly it is doing. This knowledge consists of a group of local variables for each node such as :

- the dimensions DimX, DimY, DimZ of the partition
- the total number of nodes np in the partition
- the position MyX, MyY, MyZ of the node inside the partition
- the distinctive name MyID of the node

SKIRON

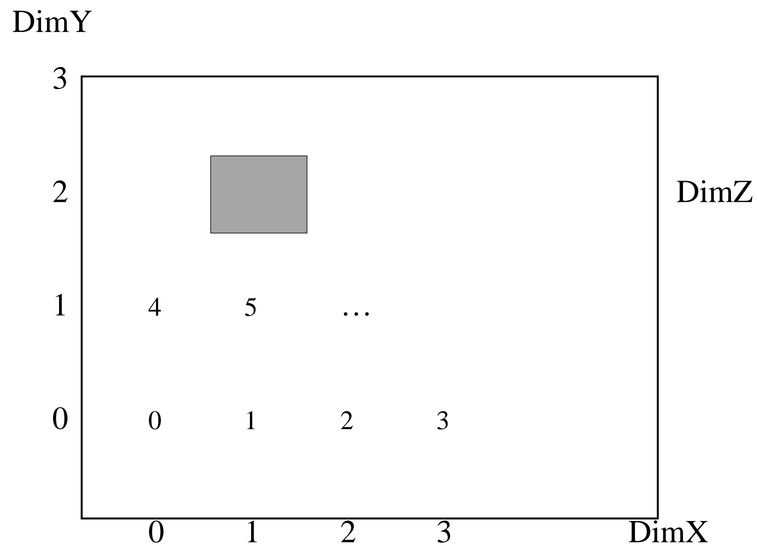


Figure 4.9.1.1.1: Partition parameters.

For the shaded node in the partition of Fig. 4.9.1.1.1 above we have

DimX = 4	DimY = 4	DimZ = 1	np = 4×4×1 = 16
MyX = 1	MyY = 2	MyZ = 0	ID = 9

9.1.2.Virtual processors

In partitions that consist solely of I/O nodes or AIX processing nodes, the number of processors on which the application will run can be defined. This is achieved by the definition, during the execution of the program, of a triad of numbers (x,y,z) which correspond to the coordinates of the processor grid that will be selected.

If the total number of processors requested is greater than the real number of available nodes, then more than one virtual processors are created in each physical

SKIRON

node.

9.2. The EPX operating system (Embedded Parix)

The environment supplied for the execution of parallel programs in the Parsytec CC system is EPX. EPX was developed with the aim of making possible the management of nodes upon which different tasks have been assigned in such a way that each node execute its task in an optimal way. The AIX operating system supports EPX with core and I/O functions.

EPX offers the capacity of creating virtual links and virtual topologies and is independent of the number of processors.

The EPX daemon (epxd) is used for the management of CC system and allows the launch of the users applications. The EPX daemon runs on all the AIX nodes (input and processing nodes). In each of the input nodes it functions as a master daemon, meaning that it waits at a port for the user's commands. When it receives a user's command, the daemon forwards it through ethernet to the slave-daemons that run in the rest of the AIX-CC nodes. The slaves will subsequently execute the command.

9.2.1.Capabilities

The EPX programming model was designed in accordance to the existing demands for parallel algorithms and combines a set of static and dynamic programming capabilities.

- Static capabilities
  - Identical main program loaded in all the processors.
  - Recognition of the network size.
  - Recognition of a specific position inside the network.
- Dynamic capabilities
  - Creation of special processes (threads)

SKIRON

- Establishment of communication lines between random processors.
- Synchronous and asynchronous communication.
- Handling of user defined virtual topologies.
- Loading and execution of additional code (contexts).

- Services
  - Use of RPC calls (Remote Procedure Calls).
  - Sustenance of specific servers.

9.2.2.Message Passing

EPX communication is based upon the principle of virtual links, which establish the connections in the grid. Three types of communication are available in EPX:

Synchronous communication through virtual links. The communicating processes must be linked by a (virtual) link and are synchronized during the communication. For example, the process which is ready to communicate with another, must wait for the second one to get ready also, and the data transmission procedure does not start until both processes are ready.

Synchronous random communication. This type of communication does not require the definition of virtual links. However, if the messages are large, library calls establish internally a virtual link for performance purposes.

Asynchronous communication through virtual links. In this case the communication is accomplished at the same time as the computations. Message transmission and receipt takes place in the background while the processor continues to run.

9.2.3.PowerPVM/EPX

SKIRON

PowerPVM/EPX is a homogeneous version of PVM specially designed for EPX/AIX which was created for the maximal exploitation of Parsytec CC system capabilities.

This PVM version is compatible with the 3.3 PVM edition and has the following distinctive features:

- PowerPVM/EPX runs exclusively on the nodes of a Parsytec CC system.
- It supplies CC systems with the capacity to exploit PVM full functionality.
- For every node the creation of up to 32 multiple dynamic processes is allowed.
- The capability of spawning selected PVM processes in specific system nodes by the use of special commands is available.
- Every parallel machine node is regarded as a host. Subsequently, a parallel machine of n nodes is considered as a network of n hosts of the same architecture.
- PVM/EPX termination comes with the exit of the last process from the parallel application.
- Power PVM/EPX offers the capacity for scalability. It has already been tested on a 40 node system and no problems are expected in the case of more nodes utilisation.

SKIRON

9.2.4.EPX features

EPX as a programming environment for CC system offers basically the following:

- Reliability
- Scalability
- High performance
- Scaling of the differences between architectures
- Independency from underlying hardware
- Ease of use

9.3. Parallel Implementation of the Eta code

The techniques used to parallelize the Eta model for the Parsytec CC-8 platform follow in general the principles described in Section 5, and especially in the case that the parallel environment of the implementation is PVM. In the case of PARIX, though, there are some differences mainly in the communication schemes that are of interest. Of course, the local communication algorithm remains the same and message passing is conducted using asynchronous send calls and synchronous receive calls. The global and mixed communication algorithms although they retain the mode of communication, they ignore the master-slave like connections of PVM and MPI, and focus on the locality of message passing at a step, taking advantage of the mesh topology.

9.3.1.Global Communication

Let *p=(px, py)* be the processor in the middle of a $dimx \times dimy$ mesh of *np* processors, i.e. for this processor *px=dimx div 2*, *py=dimy div 2*. Then, the basic idea is to progressively compute partial sums by passing the messages from each processor to a selected neighbour, until we reach processor *p*, who will receive only four sums (Instead of *np-1* which was the case of the MPI implementation) one from each neighbour and then spread the messages in reverse order. The pseudocode describing this procedure for a processor *q* is the following

```
        { SUM-all }

if  p is on my north then
{
    if  I have a down neighbour then
    {
        Receive (down, Psum1, Psum2, Psum3, Psum4)
        Sum1 = Sum1 + Psum1
        Sum2 = Sum2 + Psum2
        Sum3 = Sum3 + Psum3
```

SKIRON

```
            Sum4 = Sum4 + Psum4}
        }
        Send (upper, Sum1, Sum2, Sum3, Sum4)
        Receive (upper, Sum1, Sum2, Sum3, Sum4)

        if I have a down neighbour Send(down, Sum1, Sum2, Sum3, Sum4)
    }

    else if p is on my south then
    {
        if I have an upper neighbour then
        {
            Receive (upper, Psum1, Psum2, Psum3, Psum4)
            Sum1 = Sum1 + Psum1
            Sum2 = Sum2 + Psum2
            Sum3 = Sum3 + Psum3
            Sum4 = Sum4 + Psum4
        }
        Send     (down, Sum1, Sum2, Sum3, Sum4)
        Receive (down, Sum1, Sum2, Sum3, Sum4)

         if I have an upper neighbour then Send(upper, Sum1, Sum2, Sum3,
Sum4)
    }

    else if  I am in the same row of the mesh with p then
    {
        if  I have a down neighbour then
        {
            Receive (down, Psum1, Psum2, Psum3, Psum4)
            Sum1 = Sum1 + Psum1
            Sum2 = Sum2 + Psum2
```

SKIRON

```
        Sum3 = Sum3 + Psum3
        Sum4 = Sum4 + Psum4
}
 if I have an upper neighbour then
{
        Receive (upper, Psum1, Psum2, Psum3, Psum4)
        Sum1 = Sum1 + Psum1
        Sum2 = Sum2 + Psum2
        Sum3 = Sum3 + Psum3
        Sum4 = Sum4 + Psum4
}

if p is on my east then
{
        if I have a left neighbour then
        {
            Receive (left, Psum1, Psum2, Psum3, Psum4)
            Sum1 = Sum1 + Psum1
            Sum2 = Sum2 + Psum2
            Sum3 = Sum3 + Psum3
            Sum4 = Sum4 + Psum4
        }
        Send (right, Sum1, Sum2, Sum3, Sum4)
        Receive (right, Sum1, Sum2, Sum3, Sum4)

        if  I have a left neighbour then Send (left, Sum1, Sum2, Sum3,
        Sum4)
}

else if p is on my west then
{
        if I have a right neighbour then
```

```
        {
            Receive (right, Psum1, Psum2, Psum3, Psum4)
            Sum1 = Sum1 + Psum1
            Sum2 = Sum2 + Psum2
            Sum3 = Sum3 + Psum3
            Sum4 = Sum4 + Psum4
        }
        Send (left, Sum1, Sum2, Sum3, Sum4)
        Receive (left, Sum1, Sum2, Sum3, Sum4)

        if I have a right neighbour then Send(right, Sum1, Sum2, Sum3,
        Sum4)
}

else if I am p then
{
        if I have a left neighbour then
        Receive (left, LPsum1, LPsum2, LPsum3, LPsum4)
        if I have a right neighbour then
        Receive (right, RPsum1, RPsum2, RPsum3, RPsum4)

        Sum1 = Sum1 + LPsum1 + RPsum1
        Sum2 = Sum2 + LPsum2 + RPsum2
        Sum3 = Sum3 + LPsum3 + RPsum3
        Sum4 = Sum4 + LPsum4 + RPsum4

        if I have a left neighbour then
        Send (left, Sum1, Sum2, Sum3, Sum4)
        if I have a right neighbour then
        Send (right, Sum1, Sum2, Sum3, Sum4)
}
```

SKIRON

if I have a down neigbour then Send (down, Sum1, Sum2, Sum3, Sum4)

if I have an upper neigbour then Send (upper, Sum1, Sum2, Sum3, Sum4)

}

9.3.2.Mixed Communication

As in the case of global communication, we request $p=(px, py)$ to be a central processor of the message's route, but this time not in the center of the mesh, but in the middle of a processor column, For this reason, for a $dimx \times dimy$ mesh of *np* processors we select *dimx* processors $p_i=(px_i, py_i)$, where $0 \le px_i \le dimx-1$ and $py_i=dimy$ *div* 2. Then, the basic idea is to progressively compute partial maximums in column *i* by passing the messages from each processor either to his upper or to his down neighbour, until we reach p_i, who will receive only two maximums and then spread the messages in reverse order. The pseudocode describing this procedure for a processor *q* is the following

```
{ Max_NCLDS }

if p_i is on my north then
{
    if I have a down neighbour then
    {
        Recv (down, Pmax_i)
        Max_i = max {Max_i, Pmax_i}
    }
    Send (upper, Max_i)
Receive (upper, Max_i)


    if I have a down neighbour then Send (down, Max_i)
}
```

SKIRON

```
else if p_i is on my south then
{
    if I have an upper neighbour then
    {
        Recv (upper, Pmax_i)
        Max_i = max {Max_i, Pmax_i}
    }
    Send (down, Max_i)
    Receive (down, Max_i)

    if I have an upper neighbour then Send (upper, Max_i)
}
else if I am p_i then
{
    if I have a down neighbour then Recv (down, DPmax_i)
    if I have an upper neighbour then Recv (upper, UPmax_i)

        Max_i = max {Max_i, DPmax_i, UPmax_i}

    if I have a down neighbour then Send (down, Max_i)
    if I have an upper neighbour then Send (upper, Max_i)
}
```

9.4. Implementing communication using Parix

The most important PARIX function and procedures that were used during the design phase of the parallel implementation of the Eta model on Parsytec CC-8 are described in this section with some details concerning their arguments and their calls.

a) getdim (function)

description : returns the dimensions of the partition

call : integer DimX, DimY, DimZ

call getdim (DimX, DimY, DimZ)

b) nprocs (procedure)

description : returns the number of nodes in the partition

call : integer np

np = nprocs( )

c) getmypos (function*)*

description : returns the node's coordinates in the partition

call : integer MyX, MyY, MyZ

call getmypos(MyX, MyY, MyZ)

d) myprocid (procedure)

description : returns the id of a node

call : integer MyID

SKIRON

MyID = myprocid( )

e) newtop (procedure)

description : creates a new topology which is characterized by the number
TopId and has a maximum number of allowed links equal to
nLinks.

call : integer nLinks, TopId

TopId = newtop(nLinks)

f) addnewlink (procedure)

description : creates a new link between the calling node and the node ProcId in the topology defined by TopId, and returns the number LogLinkId that represents this link.

call : integer LogLinkId, TopId, ProcId, RequestId

LogLinkId = addnewlink(TopId, ProcId, RequestId)

SKIRON

g) px-ainit (procedure)

description : initializes all the necessary for the asynchronous communication data structures in topology TopId.

call : integer TopId, threads, size

call ainit(TopId, threads, size)

h) px-aexit (procedure)

description : frees the internal data structures used for the asynchronous communication

call : integer TopId

call aexit (TopId)

i) px-asend (procedure)

description : asynchronous send.

call : integer TopId, LogLinkId , size, result

[data_type] data

call asend(TopId, LogLinkId, data, size, result)

j) px-recv (procedure)

description : synchronous receive.

call : integer LogLinkId ,TopId, ProcId, size

[data_type] data

call recv(TopId, LogLinkId, data, size)

SKIRON

9.5.Numerical Results

The aforementioned techniques have been successfully tested and validated on the Parsytec CC platform, using PVM due to its portability, and PARIX because it was the native parallel environment of the machine.

Figures 4.9.5.2-5 represent our numerical results of the model as they were estimated for a 48h run of 1920 steps on a 121×161×32 grid, with *4*, *6* and *8* processors, respectively. Times are measured in seconds in all cases.

The reduction in time of the parallel code as compared to its sequential version is significant (Fig. 4.9.5.2). The speedup is almost linear for all versions (Fig. 4.9.5.3), and the efficiency is above *0.8* (Fig. 4.9.5.3). Fig. 4.9.5.4, 4.9.5.5 show the behaviour of the computation and communication time. PVM seems to perform better than PARIX at least when $p>6$ justifying the discrepancy identified in the speedup.

SKIRON

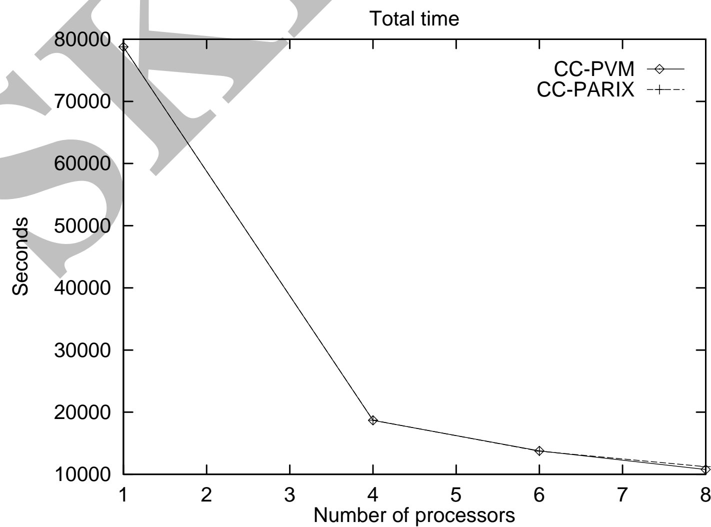


Figure 4.9.5.2: Parallel run time

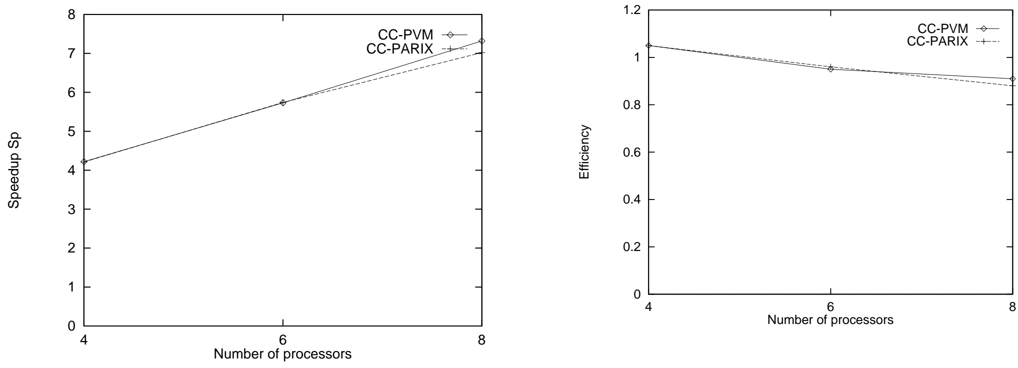


Figure 4.9.5.3: Speedup-Efficiency

SKIRON

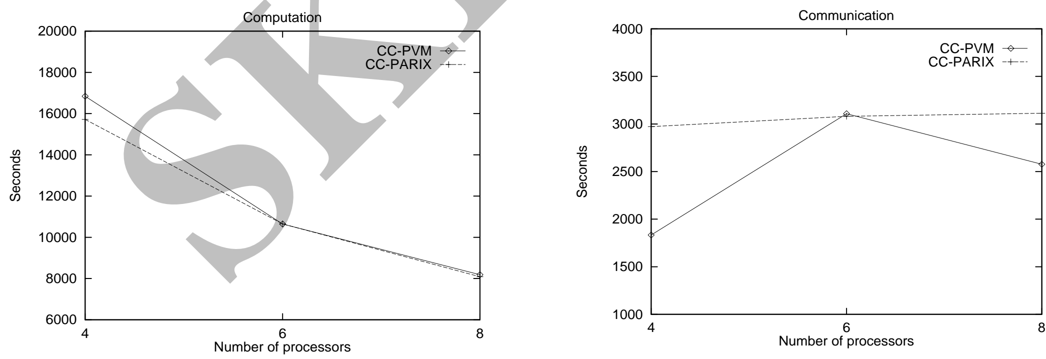


Figure 4.9.5.4: Computation-Communication

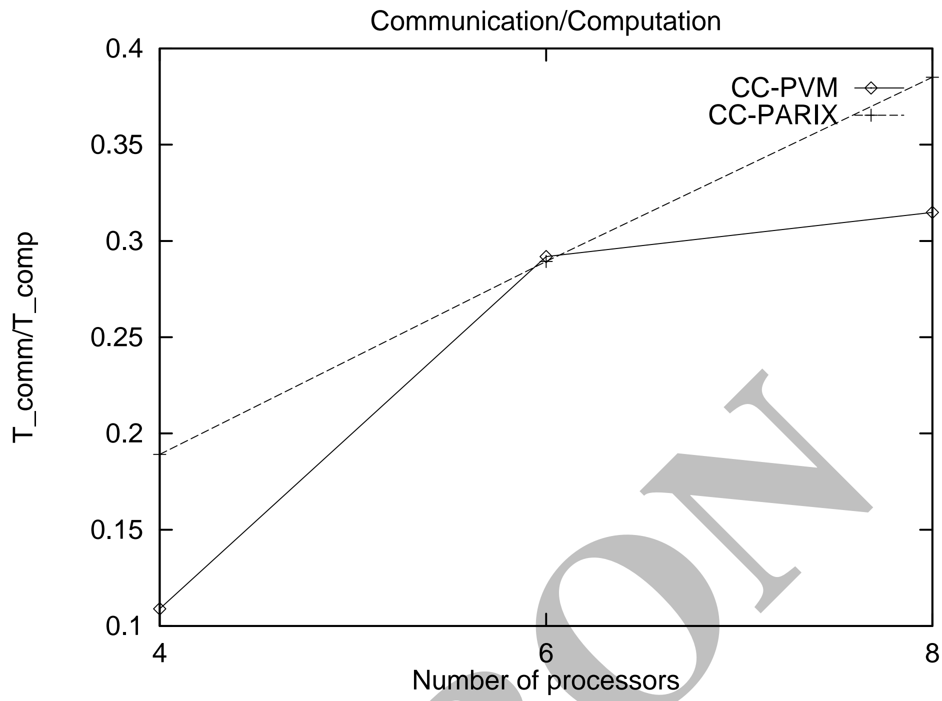


Figure 4.9.5.5: Communication/Computation

SKIRON

Appendix 4-1

Execution instructions for the parallel eta code on the Convex SPP1600 platform

SKIRON

Execution of the parallel eta code on the Convex SPP1600 platform (SKIRON Version 4.0)

1. Time selection

Directory: SKIRON.4.0/model/PAR/
Modify parameter TEND in file fcstdata.

2. Processor grid selection

Directory: SKIRON.4.0/model/SPL/
Modify parameters N_X_GRID and N_Y_GRID in file parallel.h

Example: To execute the parallel program with eight processors using a 2 x 4 mesh topology, set N_X_GRID = 2 and N_Y_GRID = 4.

3. Input file creation

NOTE: In the case that the program needs more than 30Mb memory, before the following steps, execute the command

mpa -STACK -DATA

Directory: SKIRON.4.0/model/SPL/
Data partitioning

```
make HP-UX

subsplit
sboco 1
sboco 2
  …
sboco k
```

SKIRON

where k=1(1)TEND/TBOCO

4. Execution of the parallel program

4.1. Using PVM

Directory: SKIRON.4.0/model/PAR/

Compilation

make pvm

Directory: SKIRON.4.0/modelTB/exe/

Execution

echo | pvm host_spp_pvm

peta_v4.0

echo halt | pvm

4.2. Using MPI

Directory: SKIRON.4.0/model/PAR/

Compilation

make mpi

Directory: SKIRON.4.0/model/exe/

Execution

mpirun -np n peta_v4.0

or

mpirun -np n -w peta_v4.0

where n = N_X_GRID × N_Y_GRID the total number of processes that will be executing the parallel program.

SKIRON

Example: To use a 2 x 3 mesh topology, you must have N_X_GRID, N_Y_GRID equal to 2 and 3, respectively, and start the execution of the parallel program with the command:

```
mpirun -np 6 peta_v4.0
        or
mpirun -np 6 -w peta_v4.0
```

5. Output file creation

<u>Directory</u>: *SKIRON.4.0/model/SPL/*

```
make HP-UX
submerge 000
    ….
submerge 048
```

if the parallel program was executed for a 48 hour forecast.

SKIRON

Appendix 4-2

Results of the parallel eta code on the convex SPP1600 platform with PVM and MPI

SKIRON

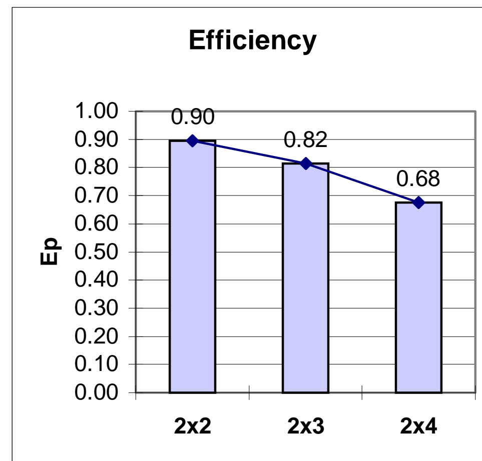
Machine SPP-1600 / PVM			
Optimization level	**O0**	**TLMOD**	**10.00**
DLMD, DPHD	**0.25, 0.25**	**TPHOD**	**40.00**
Hours	**48**	**WBD**	**-30.00**
IM X JM X LM	**121 x161 x32**	**SBD**	**-20.00**
# of iterations	**1920**	**timestep**	**90 sec**

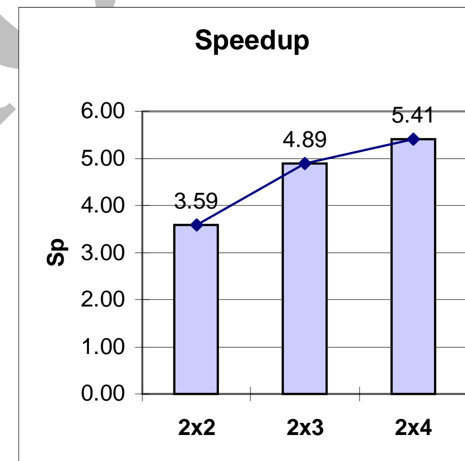
Proc. 2x2	**1**	**2**	**3**	**4**
T_seq	66959.63			
T_comp	16842.99	16364.75	16754.41	16396.34
T_comm	1833.62	2311.23	1922.21	2280.32
T_io	3.77	3.09	2.43	2.42
T_clear	18676.61	18675.98	18676.62	18676.66
T_total	18680.38	18679.07	18679.05	18679.08
Sp	3.59			
Ep	0.90			

Proc. 2x3	**1**	**2**	**3**	**4**	**5**	**6**
T_seq	66959.63					
T_comp	10989.29	11267.29	11064.74	11708.94	11107.31	11258.39
T_comm	2697.01	2417.80	2620.28	1977.80	2580.06	2429.04
T_io	5.67	5.63	5.65	3.79	3.12	3.12
T_clear	13686.30	13685.09	13685.03	13686.73	13687.37	13687.43
T_total	13691.97	13690.71	13690.67	13690.52	13690.49	13690.55
Sp	4.89					
Ep	0.82					

SKIRON

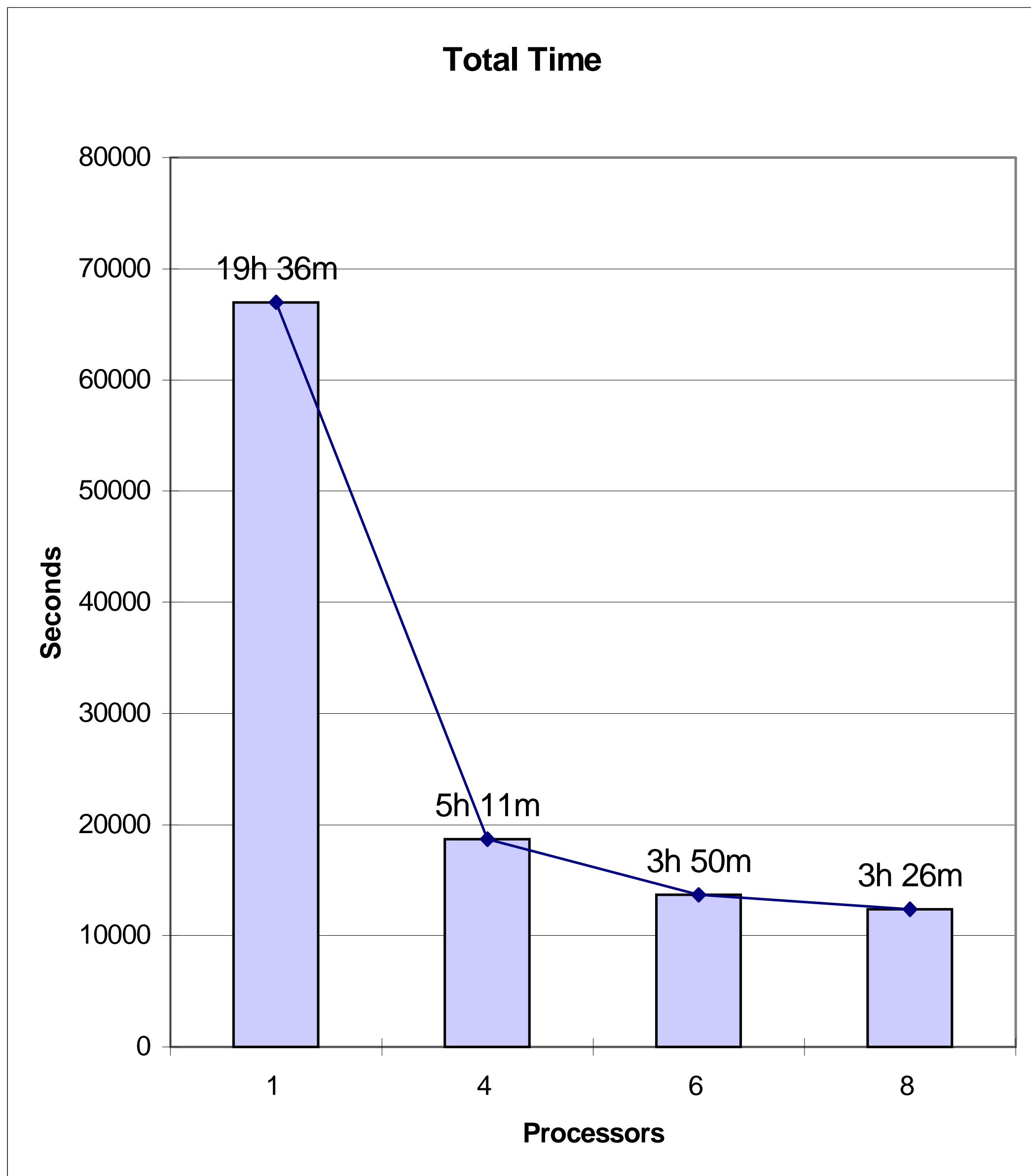
Proc. 2x4	1	2	3	4	5	6	7	8
T_seq	66959.63							
T_comp	8865.21	8887.57	8782.56	9231.32	8753.18	9545.32	8536.36	8992.63
T_comm	3514.44	3491.09	3596.13	3146.80	3625.33	2833.07	3843.15	3386.08
T_io	10.63	10.48	10.52	10.93	10.40	10.50	9.59	10.53
T_clear	12379.65	12378.65	12378.69	12378.12	12378.51	12378.39	12379.51	12378.71
T_total	12390.28	12389.13	12389.21	12389.05	12388.91	12388.89	12389.10	12389.24
Sp	5.41							
Ep	0.68							





SARON

Machine SPP-1600 / PVM			
optimization level	O0	TLMOD	10.00
DLMD, DPHD	0.25, 0.25	TPHOD	40.00
hours	48	WBD	-30.00
IM X JM X LM	121 x161 x32	SBD	-20.00
# of iterations	1920	timestep	90 sec



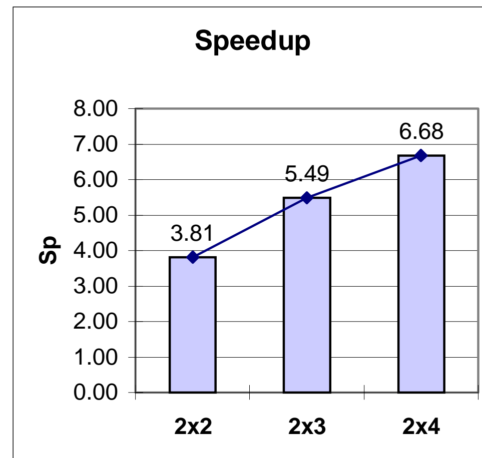
Machine SPP-1600 / MPI			
optimization level	O0	TLMOD	10,00
DLMD, DPHD	0.25, 0.25	TPHOD	40,00
hours	48	WBD	-30,00
IM X JM X LM	121 x161 x32	SBD	-20,00
# of iterations	1920	timestep	90 sec

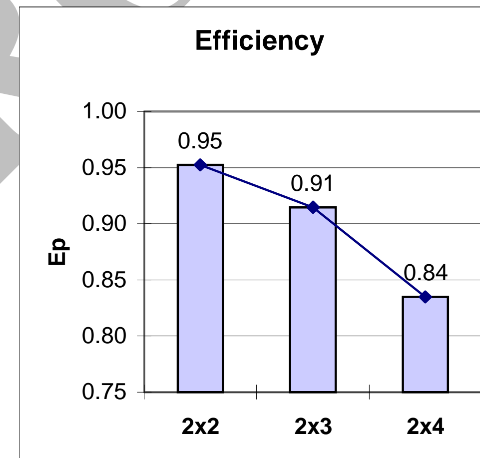
proc. 2x2	1	2	3	4
T_seq	66959,63			
T_comp	15857,85	16943,61	15694,50	16519,38
T_comm	1714,84	629,59	1876,57	1052,01
T_io	2,18	1,66	3,75	3,61
T_clear	17572,69	17573,19	17571,08	17571,39
T_total	17574,87	17574,86	17574,83	17575,00
Sp	3,81			
Ep	0,95			

proc. 2x3	1	2	3	4	5	6
T_seq	66959,63					
T_comp	10769,42	11121,40	11000,81	11421,39	10684,04	11086,43
T_comm	1424,27	1072,82	1193,51	774,70	1510,55	1108,83
T_io	5,76	5,17	5,09	3,30	4,75	4,37
T_clear	12193,70	12194,22	12194,32	12196,09	12194,59	12195,25
T_total	12199,46	12199,39	12199,41	12199,40	12199,34	12199,63
Sp	5,49					
Ep	0,91					

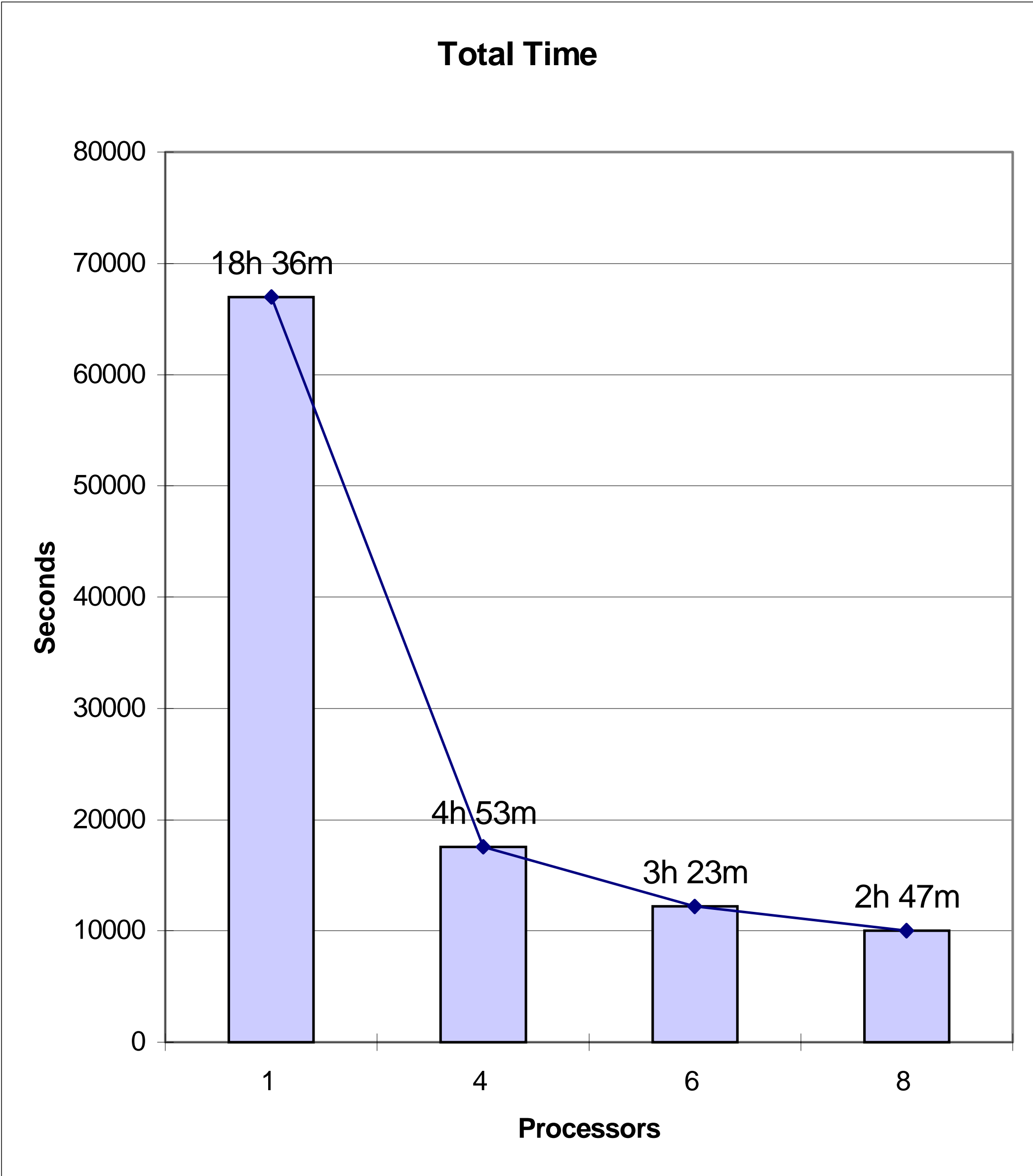
SKIRON

proc. 2x4	1	2	3	4	5	6	7	8
T_seq	66959,63							
T_comp	8318,69	8583,78	8464,25	8839,55	8467,24	9180,78	8187,24	8505,67
T_comm	1702,82	1437,79	1557,35	1182,01	1554,74	841,13	1834,57	1516,04
T_io	2,08	2,02	1,93	1,91	1,43	1,49	1,58	2,05
T_clear	10021,51	10021,57	10021,61	10021,56	10021,97	10021,92	10021,81	10021,70
T_total	10023,58	10023,59	10023,53	10023,47	10023,40	10023,41	10023,39	10023,75
Sp	6,68							
Ep	0,84							





Machine SPP-1600 / MPI			
optimization level	**O0**	**TLMOD**	**10,00**
DLMD, DPHD	**0.25, 0.25**	**TPHOD**	**40,00**
hours	**48**	**WBD**	**-30,00**
IM X JM X LM	**121 x161 x32**	**SBD**	**-20,00**
# of iterations	**1920**	**timestep**	**90 sec**

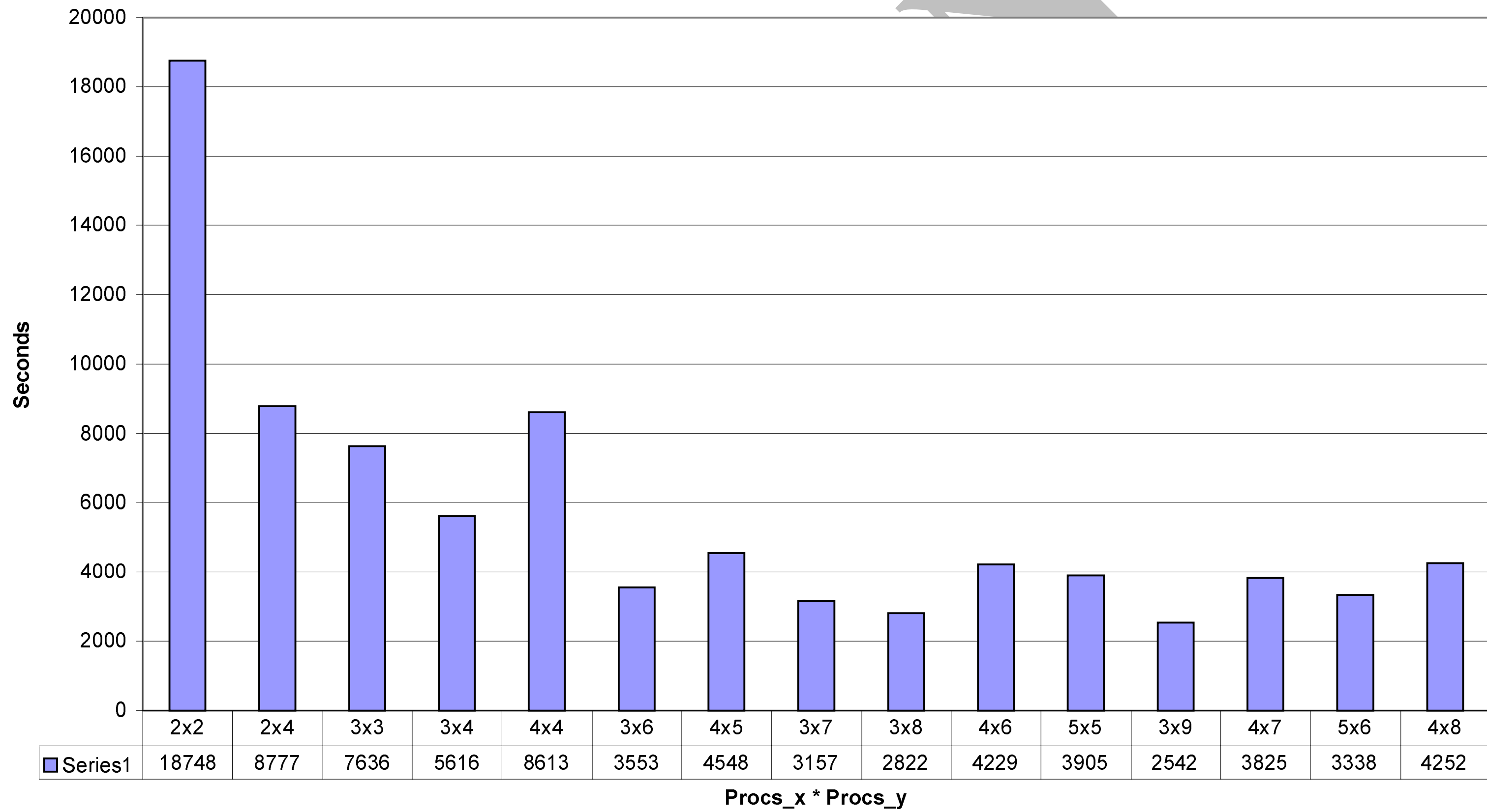


Appendix 4-3

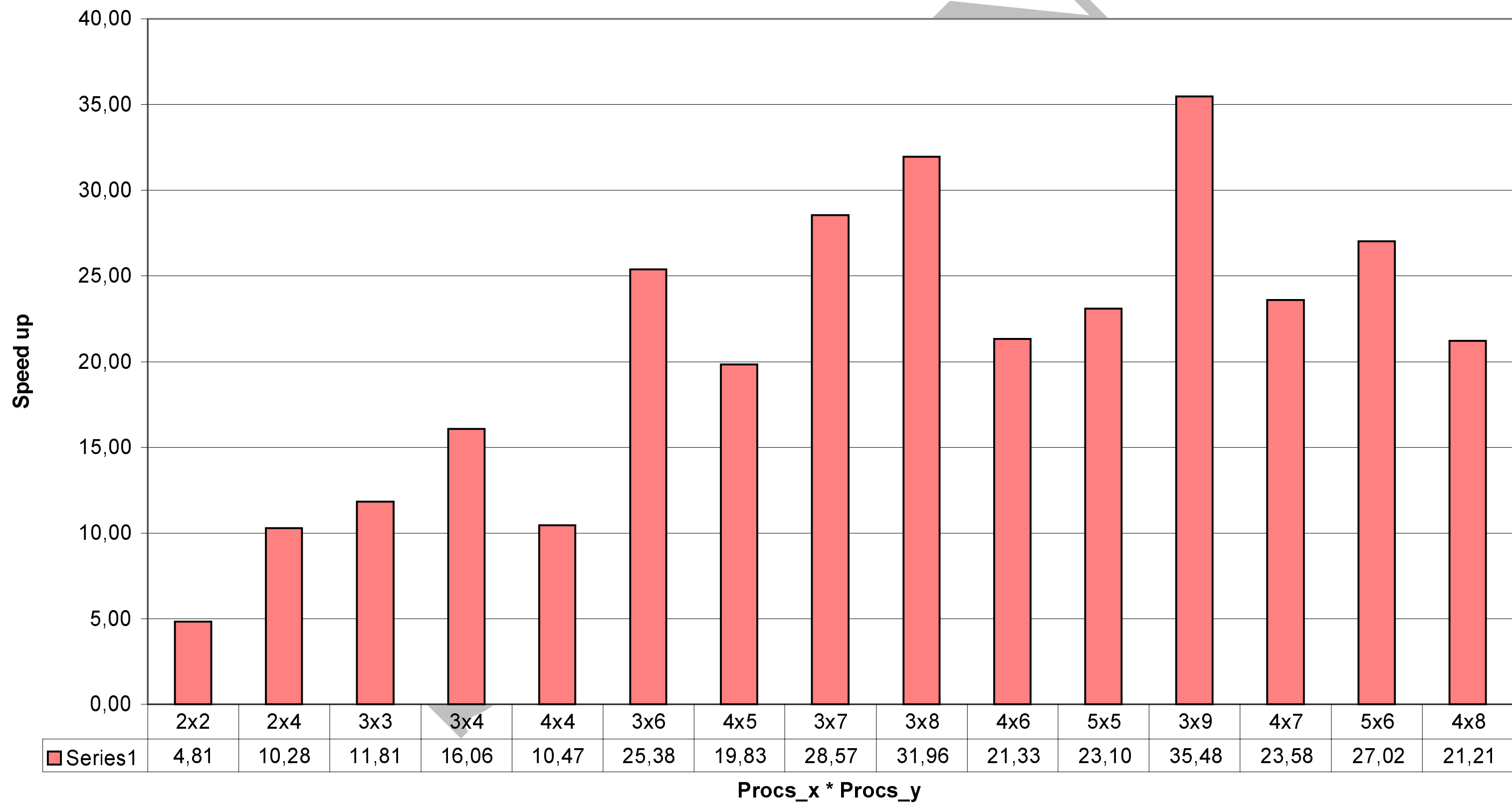
Results of the parallel eta code on the Convex SPP-2000 platform with MPI

SKIRON

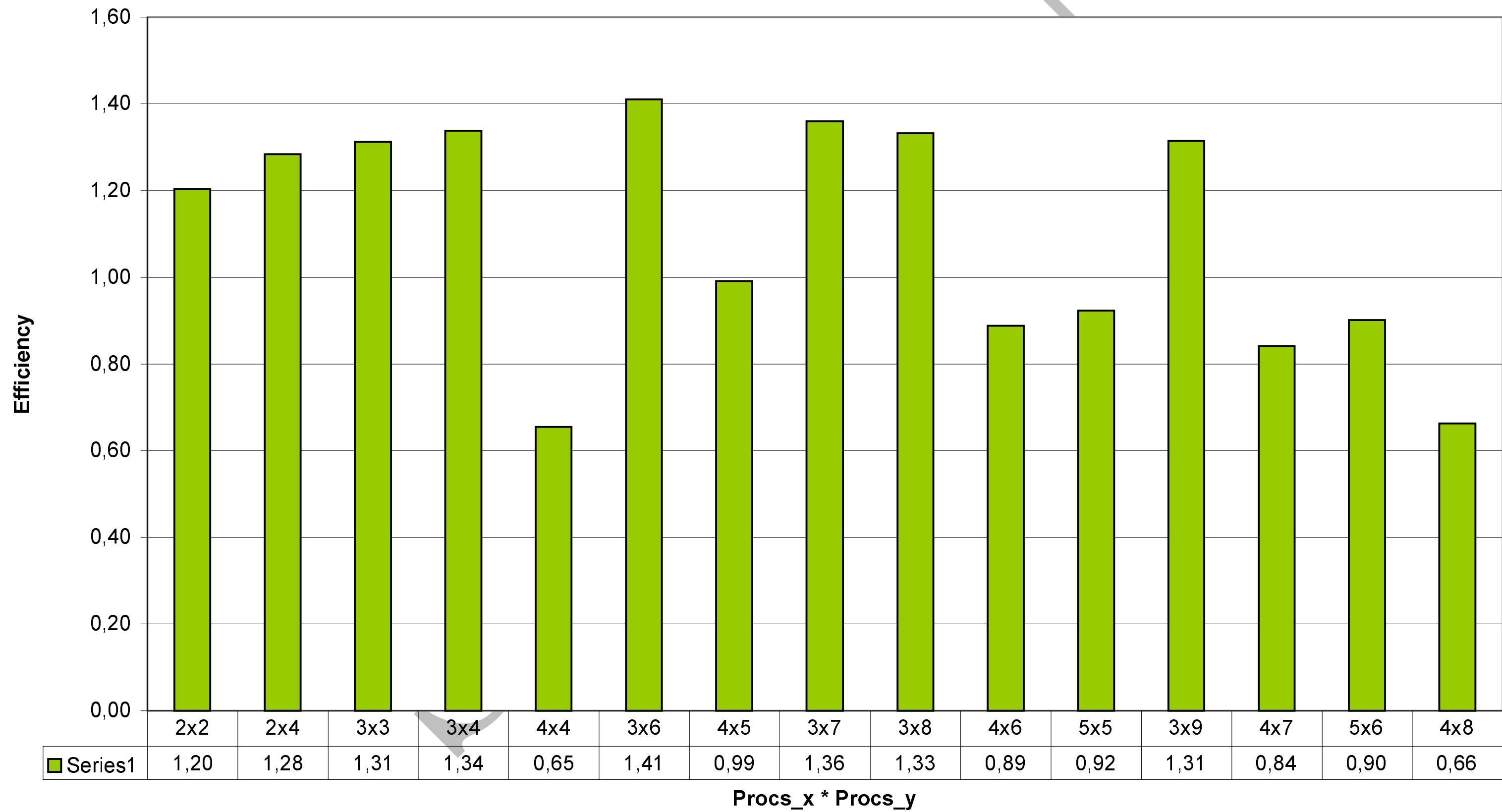
Machine	*Convex SPP-2000 /MPI*		
Optimization level	**O2**	**TLMOD**	**10,00**
DLMD, DPHD	**0.125, 0.125**	**TPHOD**	**40,00**
Hours	**12**	**WBD**	**-30,00**
IM X JM X LM	**241 x321 x32**	**SBD**	**-20,00**
# of iterations	**960**	**Timestep**	**45 sec**



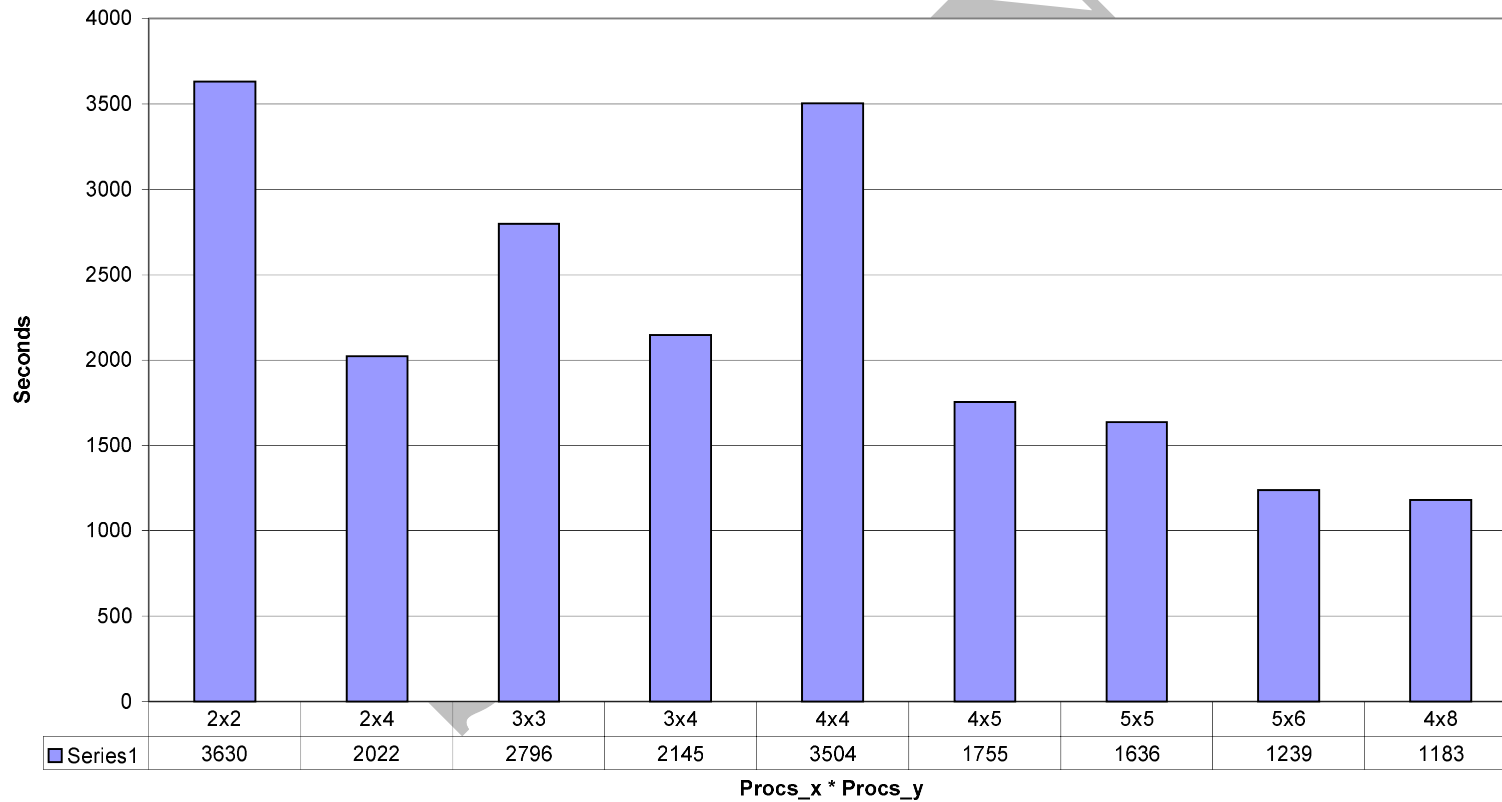
Machine	*Convex SPP-2000 /MPI*		
Optimization level	**O2**	**TLMOD**	**10,00**
DLMD, DPHD	**0.125, 0.125**	**TPHOD**	**40,00**
Hours	**12**	**WBD**	**-30,00**
IM X JM X LM	**241 x321 x32**	**SBD**	**-20,00**
# of iterations	**960**	**Timestep**	**45 sec**



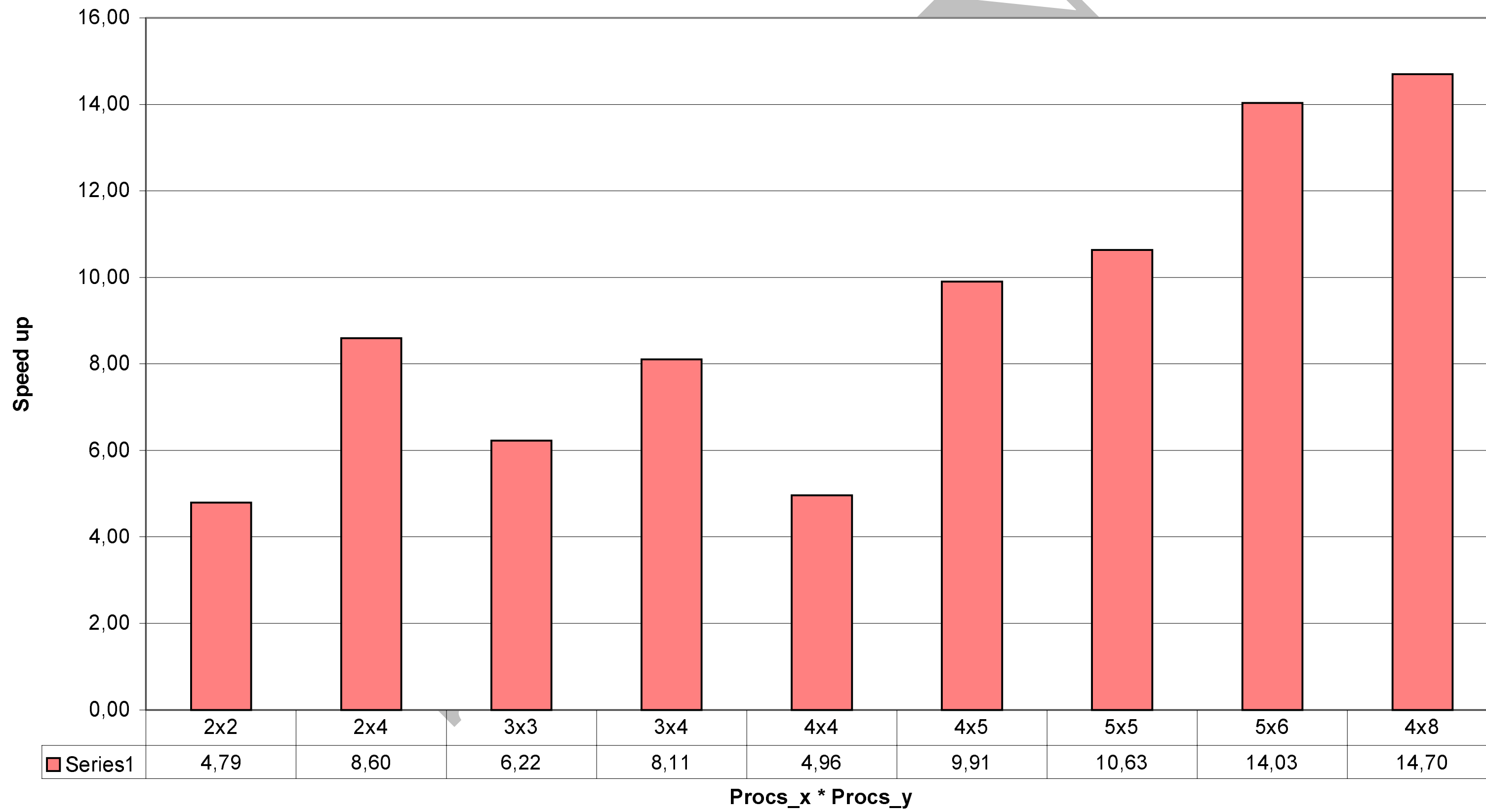
Machine	*Convex SPP-2000 /MPI*		
Optimization level	**O2**	**TLMOD**	**10,00**
DLMD, DPHD	**0.125, 0.125**	**TPHOD**	**40,00**
Hours	**12**	**WBD**	**-30,00**
IM X JM X LM	**241 x321 x32**	**SBD**	**-20,00**
# of iterations	**960**	**Timestep**	**45 sec**



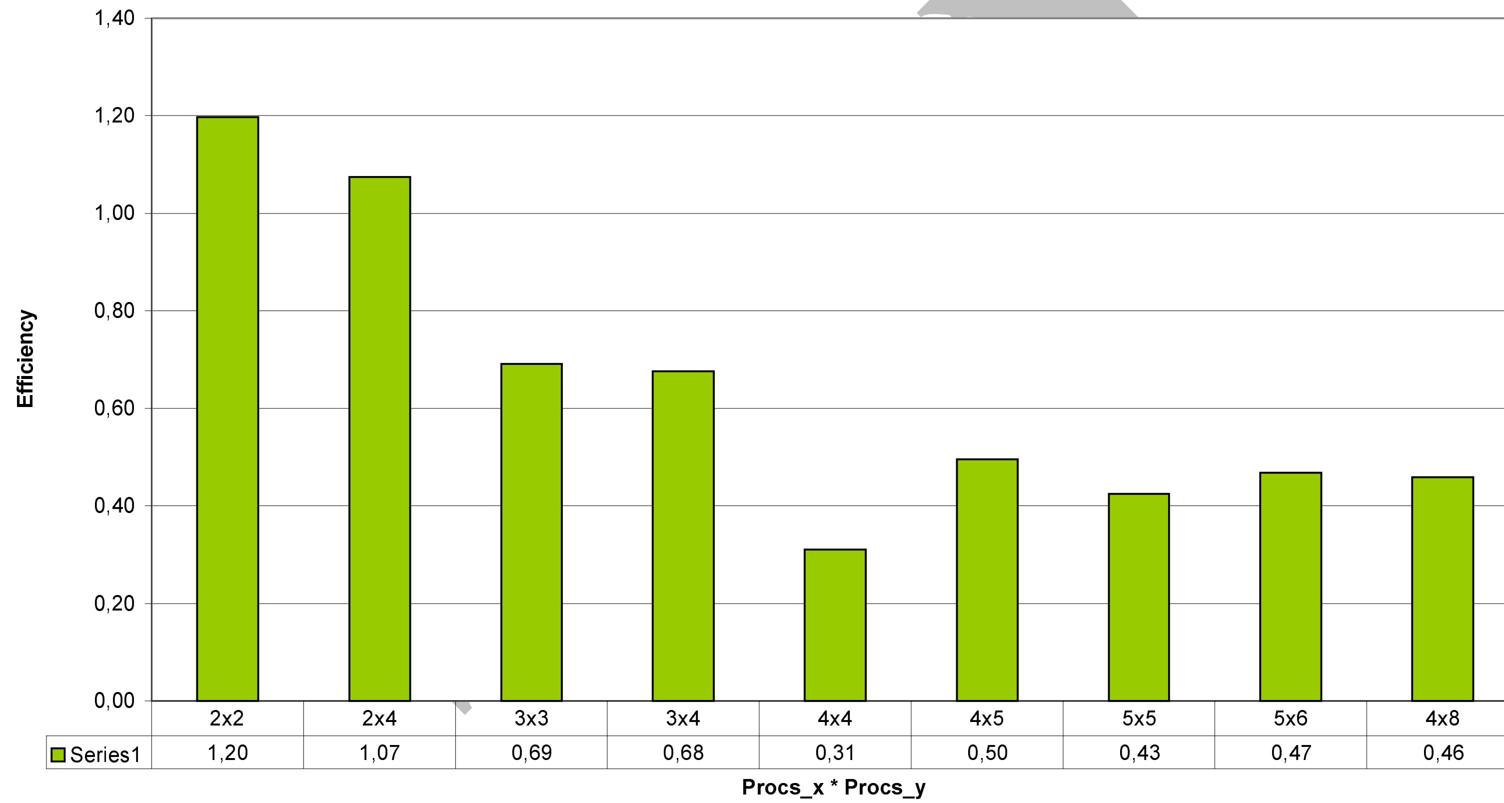
Machine	Convex SPP-2000 /MPI		
Optimization level	O2	TLMOD	10,00
DLMD, DPHD	0.25, 0.25	TPHOD	40,00
Hours	24	WBD	-30,00
IM X JM X LM	121 x 161 x 32	SBD	-20,00
# of iterations	960	Timestep	90 sec



Machine	*Convex SPP-2000 /MPI*		
Optimization level	**O2**	**TLMOD**	**10,00**
DLMD, DPHD	**0.25, 0.25**	**TPHOD**	**40,00**
Hours	**24**	**WBD**	**-30,00**
IM X JM X LM	**121 x 161 x 32**	**SBD**	**-20,00**
# of iterations	**960**	**Timestep**	**90 sec**



Machine	*Convex SPP-2000 /MPI*		
Optimization level	**O2**	**TLMOD**	**10,00**
DLMD, DPHD	**0.25, 0.25**	**TPHOD**	**40,00**
Hours	**24**	**WBD**	**-30,00**
IM X JM X LM	**121 x 161 x 32**	**SBD**	**-20,00**
# of iterations	**960**	**Timestep**	**90 sec**



Appendix 4-4

Results of the parallel eta code on the Parsytec CC-8 platform with PARIX and PVM

SKIRON

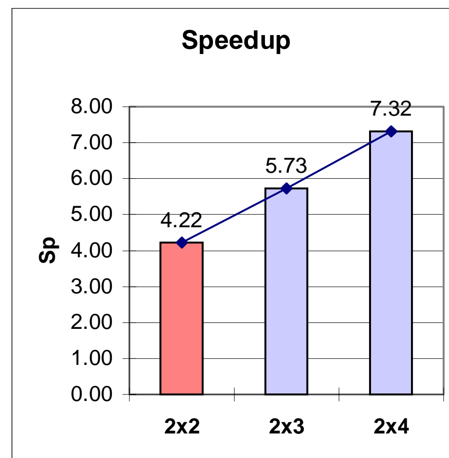
machine	*PARSYTEC CC / PVM*		
optimization level	**O3**	**TLMOD**	**10,00**
DLMD, DPHD	**0.25, 0.25**	**TPHOD**	**40,00**
hours	**48**	**WBD**	**-30,00**
IM X JM X LM	**121 x161 x32**	**SBD**	**-20,00**
# of iterations	**1920**	**timestep**	**90 sec**

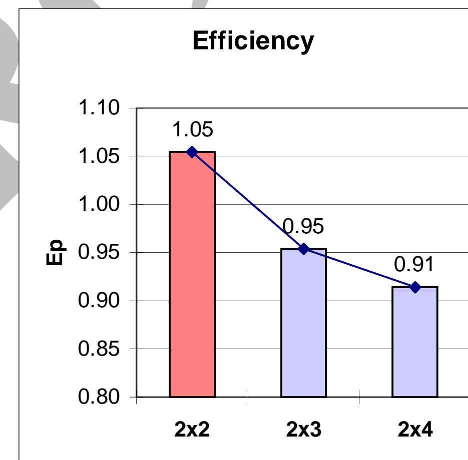
proc. 2x2	**1**	**2**	**3**	**4**
T_seq	78781,76			
T_comp	16842,99	16364,75	16754,41	16396,34
T_comm	1833,62	2311,23	1922,21	2280,32
T_io	3,77	3,09	2,43	2,42
T_clear	18676,61	18675,98	18676,62	18676,66
T_total	18680,38	18679,07	18679,05	18679,08
Sp	4,22			
Ep	1,05			

proc. 2x3	**1**	**2**	**3**	**4**	**5**	**6**
T_seq	78781,76					
T_comp	10646,36	10872,78	11048,88	11738,09	10346,86	10846,16
T_comm	3107,75	2879,62	2703,23	2013,60	3404,73	2905,40
T_io	11,49	11,49	11,48	11,48	11,48	11,48
T_clear	13754,11	13752,40	13752,11	13751,69	13751,59	13751,56
T_total	13765,60	13763,89	13763,59	13763,17	13763,07	13763,04
Sp	5,73					
Ep	0,95					

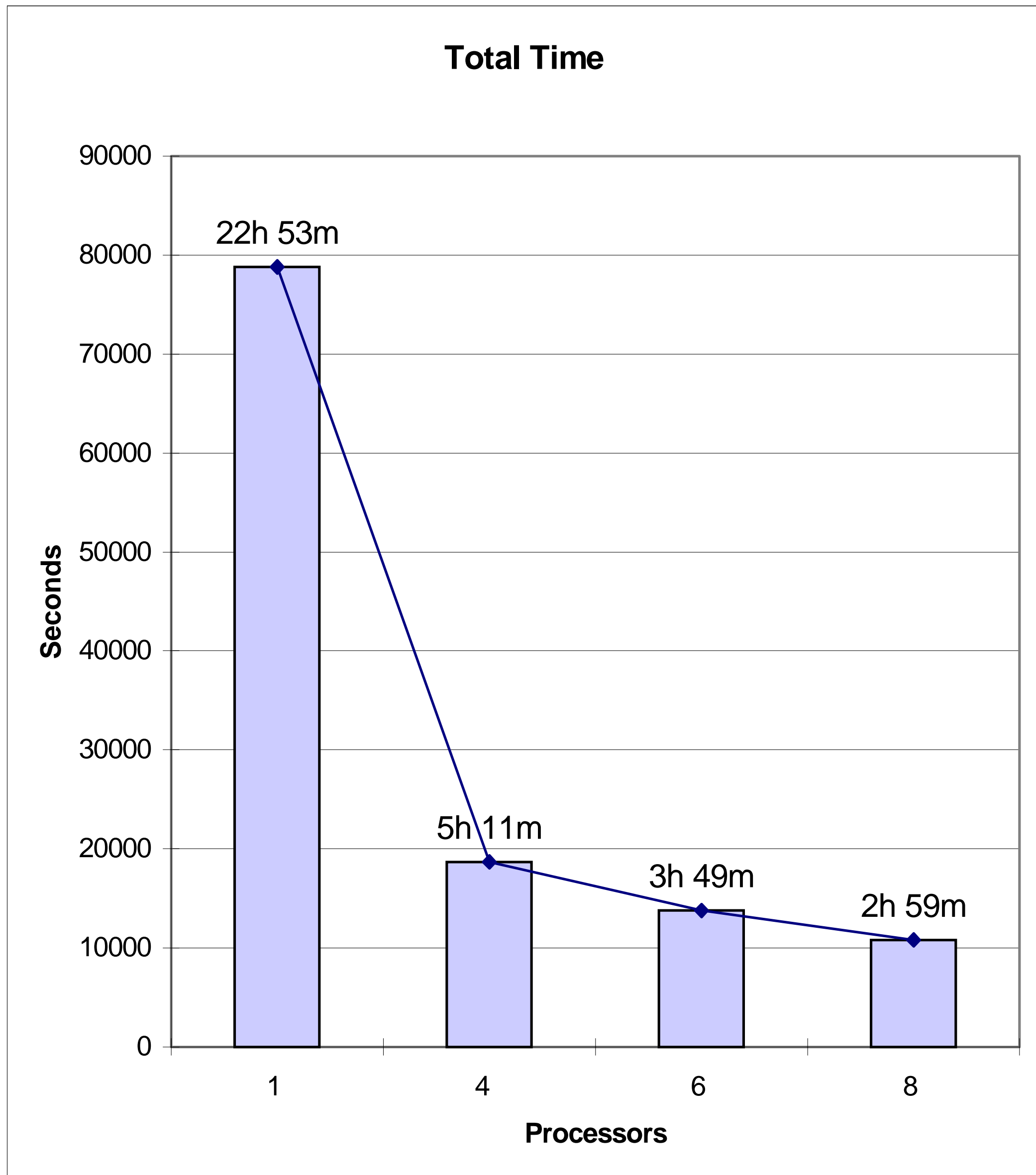
SKIRON

proc. 2x4	1	2	3	4	5	6	7	8
T_seq	78781,76							
T_comp	8186,80	8230,85	8034,74	8344,52	8009,79	8576,42	8209,18	8548,76
T_comm	2577,01	2531,32	2726,79	2416,55	2751,25	2184,46	2551,26	2211,49
T_io	12,83	12,83	12,83	12,83	12,82	12,82	12,82	12,82
T_clear	10763,81	10762,17	10761,53	10761,08	10761,04	10760,87	10760,44	10760,25
T_total	10776,64	10775,00	10774,36	10773,90	10773,86	10773,69	10773,26	10773,06
Sp	7,32							
Ep	0,91							





	machine	*PARSYTEC CC / PVM*		
optimization level	**O3**		**TLMOD**	**10,00**
DLMD, DPHD	**0.25, 0.25**		**TPHOD**	**40,00**
hours	**48**		**WBD**	**-30,00**
IM X JM X LM	**121 x161 x32**		**SBD**	**-20,00**
# of iterations	**1920**		**timestep**	**90 sec**



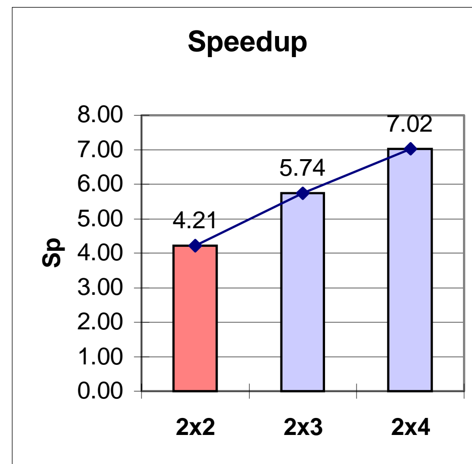
machine	*PARSYTEC CC / PARIX*		
optimization level	**O3**	**TLMOD**	**10,00**
DLMD, DPHD	**0.25, 0.25**	**TPHOD**	**40,00**
hours	**48**	**WBD**	**-30,00**
IM X JM X LM	**121 x161 x32**	**SBD**	**-20,00**
# of iterations	**1920**	**timestep**	**90 sec**

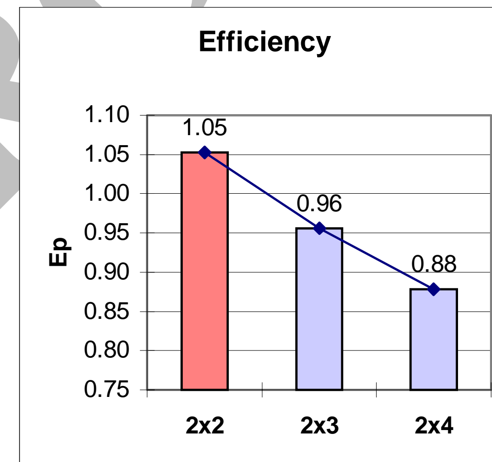
proc. 2x2	**1**	**2**	**3**	**4**
T_seq	78781,76			
T_comp	15720,39	16251,03	16080,62	16980,45
T_comm	2972,44	2441,46	2612,05	1712,17
T_io	10,60	10,48	10,44	10,08
T_clear	18692,83	18692,49	18692,67	18692,62
T_total	18703,44	18702,97	18703,11	18702,71
Sp	4,21			
Ep	1,05			

proc. 2x3	**1**	**2**	**3**	**4**	**5**	**6**
T_seq	78781,76					
T_comp	10648,25	10795,54	10935,86	11575,09	10391,54	10774,10
T_comm	3080,12	2932,56	2792,37	2153,11	3336,82	2954,11
T_io	11,59	11,29	11,23	11,23	11,24	11,23
T_clear	13728,37	13728,10	13728,23	13728,20	13728,37	13728,21
T_total	13739,96	13739,39	13739,46	13739,42	13739,61	13739,45
Sp	5,74					
Ep	0,96					

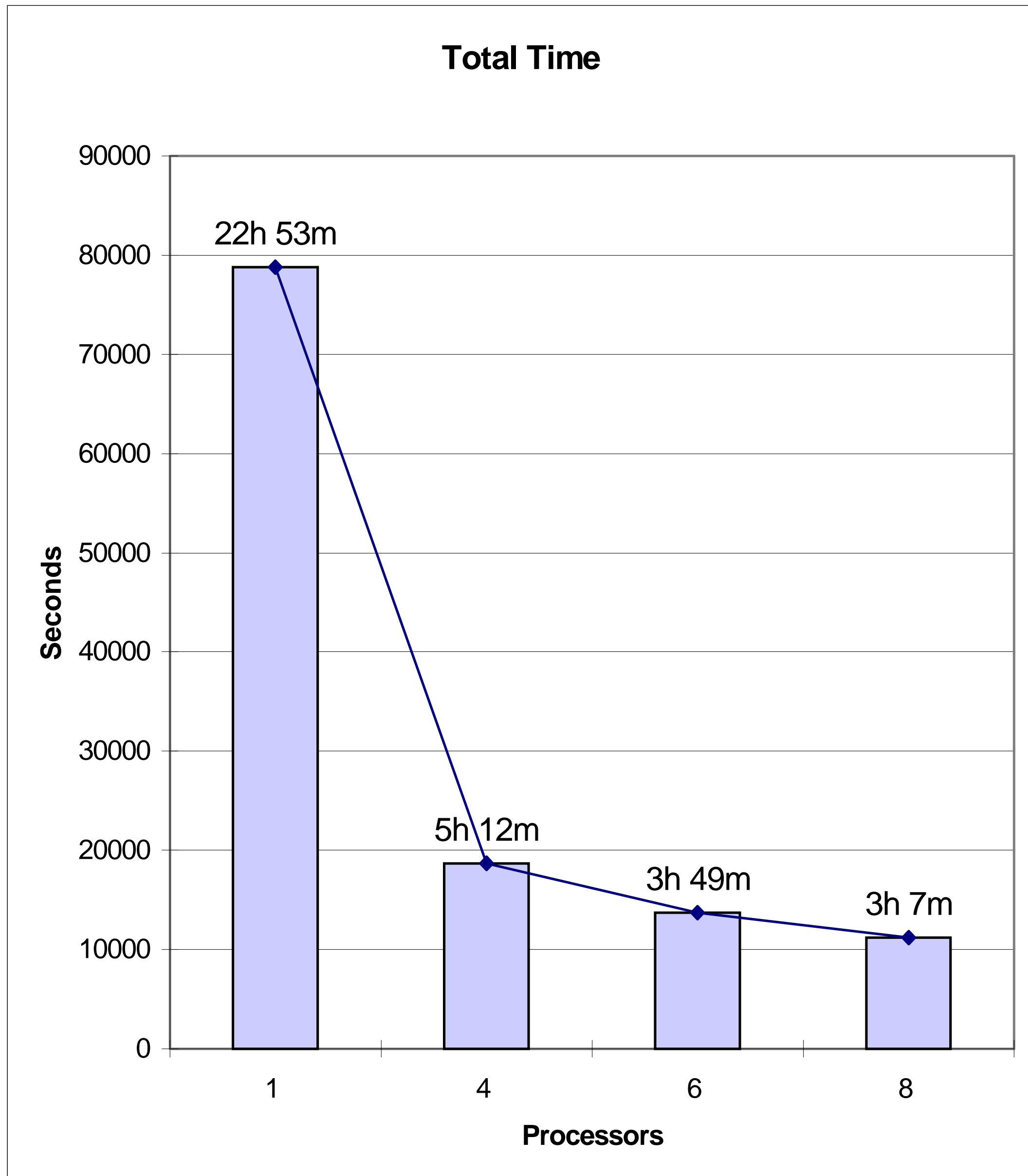
SKIRON

proc. 2x4	1	2	3	4	5	6	7	8
T_seq	78781,76							
T_comp	8087,34	8164,04	8235,20	8694,99	8016,12	8471,75	7808,17	8137,91
T_comm	3113,40	3036,49	2965,43	2505,62	3184,62	2728,87	3392,40	3062,87
T_io	15,18	12,18	13,40	13,21	12,66	13,46	14,35	11,75
T_clear	11200,73	11212,71	11214,03	11213,81	11213,41	11214,07	11214,92	11212,52
T_total	11215,92	11212,71	11214,03	11213,81	11213,41	11214,07	11214,92	11212,52
Sp	7,02							
Ep	0,88							





	Machine	*PARSYTEC CC / PARIX*	
Optimization level	**O3**	**TLMOD**	**10,00**
DLMD, DPHD	**0.25, 0.25**	**TPHOD**	**40,00**
Hours	**48**	**WBD**	**-30,00**
IM X JM X LM	**121 x161 x32**	**SBD**	**-20,00**
# of iterations	**1920**	**Timestep**	**90 sec**



References

Boukas L.A., Mimikou N.Th., Missirlis N.M., 1997: A Distributed Implementation of the Numerical Weather Prediction Eta Model. Presented at the *IASTED International Conference on Parallel and Distributed Systems, Euro PDS'97*, June 9-11, Barcelona, Spain and appeared in the *Proc. of the IASTED Conference on Parallel and Distributed Computing and Networks*, *IASTED/Acta Press*, 301-304 (also accepted, after selection, to be published in the *IASTED Journal for Parallel and Distributed Systems*).

Convex Computer Corporation, 1995: MPI: A Message-Passing Interface Standard. Message Passing Interface Forum.

Dongarra J., Geist G., Manchek R., Sunderam V., 1993: Integrated PVM Framework Supports Heterogeneous Network Computing. ORNL.

Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R., Sunderam V., 1994: PVM: Parallel Virtual Machine. A user's Guide and Tutorial for Networked Parallel Computing. Scientific and Engineering Computation Series, Massachusetts Institute of Technology.

Geist A., Beguelin A., Dongarra J., Jiang W., Manchek R., Sunderam V., 1994: PVM3 User's guide and Reference manual, Engineering Physics and Mathematics Division, Mathematical Sciences Section, ORNL/TM-12187.

Genias Software GmbH, 1996: PowerPVM/EPX: Parallel Virtual Machine for Parsytec CC Systems under EPX/AIX.

Gropp W., Doss N, Skjellum A., 1996: MPICH Model MPI Implementation Reference Manual.

Henderson T., Baillie C., Benjamin S., Govett M., Hart L., Marroquin A., Rodriguez B., 1995: Progress Toward Demonstrating Operational Capability of Massively Parallel Processors at the Forecast Systems Laboratory, in

SKIRON

Coming of Age. *Proc. of the sixth ECMWF Workshop on the use of Parallel Processors in Meteorology*, *World Scientific*.

Hewlett-Packard Company, 1996: Convex PVM/GSM User's Guide for Exemplar Systems. Convex Technology Center, Order No. DSW-501, Third Edition.

Hewlett-Packard Company, 1996: Exemplar Programming Guide. Convex Technology Center, Order No. DSW-067, Third Edition.

Hewlett-Packard Company, 1996: Exemplar SPP1600-Series Architecture. Convex Technology Center, Order No. DHW-014.

Hewlett-Packard Company, 1996: MPICH User's Guide for Exemplar Systems. Convex Technology Center, Order No. DSW-493, Second Edition.

Janjic Z.I., 1979: Forward-backward scheme modified to prevent two-grid-interval noise and its application in sigma coordinate models. *Contrib. Atmos. Phys.*, **52**, 69-84.

Janjic Z.I., 1984: Non-linear advection schemes and energy cascade on semi-staggered grids. *Mon. Wea. Rev.*, **112**, 1234-1245.

Janjic Z.I., 1990: The step-mountain coordinate: physical package. *Monthly Weather Review*, **118**, 1429-1443.

Janjic Z.I., 1994: The Step-mountain Eta Coordinate Model: Further Developments of the Convection Viscous Sublayer and Turbulence Closure Schemes. *Monthly Weather Review*, **1-2**, 927-945.

Janjic Z.I., and Mesinger F., 1984: Finite-difference methods for the shallow water equations on various horizontal grids. *Numerical Methods for Weather Prediction*, *Seminar 1983*, ECMWF, Reading, U.K, **1**, 29-101.

SKIRON

Kallos G., 1997: The Regional Weather Forecasting System SKIRON: A General Overview, *Proc. of the Symposium on Regional Weather Prediction on Parallel Computer Environments*, Athens, Greece.

Kallos G., Nickovic S., Jovic D., Kakaliagou O., Papadopoulos A., Missirlis N., Boukas L. and Mimikou N., 1997: The Eta Model Operational Forecasting System and its Parallel Implementation. *1st Workshop on Large-Scale Scientific Computations*, Varna, Bulgaria.

Mesinger F., 1973: A method for construction of second-order accuracy difference schemes permitting no false two-grid-interval wave in the height field. *Tellus*, **25**, 444-458.

Mesinger F., 1976: An economical explicit scheme which inherently prevents the false two-grid-interval wave in the forecast fields. *Proc. Symp. on Difference and Spectral Methods for Atmosphere and Ocean Dynamic Problems*, *Novosibirsk*, 17-22 September 1973, Acad. Sci., Novosibirsk, **Part II**, 18-34.

Mesinger F., 1984: A blocking technique for representation of mountains in atmospheric models. *Rivista di Meteorologia Aeronautica*, **44**, No. 1-4, 195-202.

Mesinger F., 1997: Forward-backward scheme, and its use in a limited area model. *Contrib. Atmos. Phys.*, **50**, 200-210.

Mesinger F., and Arakawa A., 1976: Numerical methods used in atmospheric models. *GARP Publication Series*, **1**, No. 17, WMO-ICSU Joint Organizing Committee, Geneva.

Mesinger F., Janjic Z.I., Nickovic S., Gavrilov D. and Deaven D.G., 1988: The step-mountain coordinate: Model description and performance for cases

SKIRON

of Alpine lee cyclogenesis and for a case of an Appalachian redevelopment. *Mon. Wea. Rev.*, **116**, 1493-1518.

Ortega J.M., 1988: Introduction to Parallel and Vector Solution of Linear Systems, *Plenum Press*.

Ritchie H., Temperton C., Simmons A., Hortal M., Davies D. and Hamrud M., 1995: Implementation of the semi-Lagrangian method in a high resolution of the ECMWF forecast model. *Monthly Weather Review*, **123**, 489-514.

Sunderam V., Geist G., Dongarra J. & Manchek R.: "The PVM Concurrent Computing System: Evolution, Experiences and Trends", ORNL.

SKIRON