PARALLELIZING OPERATIONAL WEATHER FORECAST MODELS FOR PORTABLE AND FAST EXECUTION

Bernardo Rodriguez, Leslie Hart, and Tom Henderson

High Performance Computing Group

NOAA Forecast Systems Laboratory

Title: Parallelizing Operational Weather Forecast Models

Corresponding author: Bernardo Rodriguez

Address: P.O.Box 7115, Boulder, CO 80306 USA

Phone number: (303) 497-6096

E-mail: bernardo@fsl.noaa.gov

Abstract. This paper describes a high level library (The Nearest Neighbor Tool: NNT) that has been used to parallelize operational weather prediction models. NNT is part of the Scalable Modeling System (SMS), developed at the Forecast Systems Laboratory (FSL). Programs written in NNT rely on SMS's run-time system and port between a wide range of computing platforms, performing well in multiprocessor systems. We show, using examples from operational weather models, how large Fortran 77 codes can be parallelized using NNT. We compare the ease of programmability of NNT and High Performance Fortran (HPF). We also discuss optimizations like data movement overlap (in interprocessor communication and I/O operations), and the minimization of data exchanges through the use of redundant computations. We show that although HPF provides a simpler programming interface, NNT allows for program optimizations that increase performance considerably and still keeps a simple user interface. These optimizations have proven essential to run weather prediction models in real time, and HPF compilers should incorporate them in order to meet operational demands. Throughout the paper we present performance results of weather models running on a network of workstations, the Intel Paragon, and the SGI Challenge. Finally, we study the cost of programming global address space architectures with NNT's local address space paradigm.

BIOGRAPHIES

BERNARDO RODRIGUEZ is a Senior Consultant with the Network Solutions Group at Andersen Conculting. He worked during five years as a Research Associate at NOAA's Forecast Systems Laboratory, Boulder, Colorado. He obtained his Bachelor of Science Degree from the Universidad Rafael Urdaneta, in Maracaibo, Venezuela, in 1987; and a Master of Science and a Ph.D.in Electrical and Computer Engineering at the University of Colorado at Boulder, in 1989 and 1995 respectively. His research interests are computer architecture and broadband communication systems.

TOM HENDERSON received a Bachelor's degree in Applied Physics from the University of California at San Diego in 1984 and a M.S. degree in Electrical Engineering from San Diego State University in 1990. He is currently a computer scientist with Science and Technology Corporation under contract to NOAA Forecast Systems Laboratory, Boulder, Colorado. His research interests include high level libraries for parallelizing large scientific applications, high-performance parallel I/O, and parallel languages and tools.

LESLIE HART received his M.S. in Applied Mathematics from the University of Colorado, Denver, 1986. He is currently a computer scientist with Science and Technology Corporation under contract to NOAA Forecast Systems Laboratory, Boulder, Colorado. His research interests are high level libraries, parallel I/O, run-time systems, and parallel tools.

1. INTRODUCTION

Operational weather forecast offices, like the National Center for Environmental Prediction (NCEP), run numerical weather simulation programs to produce daily forecasts. These programs require very high computation and I/O speeds. Massively parallel computers are being investigated at NOAA's Forecast Systems Laboratory as a way of meeting current and future demands [3]. Besides performance requirements, "soft" requirements are also very important, such as portability, reduced development cost, and programmability. It is necessary that the code and data files port among computing platforms (sequential and parallel) without source changes so that new machines can be used and research can be conducted on different platforms. It is also important to maintain a single version of the code. The transfer from current sequential programs (sometimes with directives for vector Supercomputers, like Crays) to the new programs that execute on parallel and sequential architectures should occur with minimal cost. Moreover, the changes to the code should have a minimum impact on the code "appearance," so that model developers can recognize and modify sections of code in the new program.

We have designed NNT to be used in the development of codes that comply with the requirements described above. NNT is part of the SMS, which has another component that makes the run-time system. NNT provides a local address space programming environment. It is a set of Fortran 77 callable routines built over a layer of lower level routines that form the interface with the specific computing platform where the codes must be run. NNT has been ported to single processor computing environments, such as workstations and single-processor Crays, and multiple-processor computing environments like the Intel Paragon, SGI Challenge, Cray T3D, IBM SP2, nCUBE , and workstation clusters. A port to the MPI message passing standard [12] is under way. Programs written exclusively using NNT and Fortran 77 port among platforms without any source code change, as do the input and output data files. Machine dependent code, like explicit message passing or synchronization operations, may be used with NNT but inhibit portability. NNT allows for optimization in two areas: at the user level and at the implementation level. At the user level, NNT provides routines for the run-

time decomposition of data to processors, the overlap of computation and communication, and the overlap of processor activities with I/O operations. These are described in Section 3. The overlap of communication, I/O, and computation, is managed by SMS's run-time system. At the implementation level, NNT allows machine-specific optimizations that are invisible to the user. These are usually related to message passing, I/O, global or local address space architectures, and single or multiple process execution. For example, NNT implementations may use machine-dependent I/O calls that enhance I/O throughput, and avoid multilevel buffering in machines with global address space.

As a sequential code is modified for parallel execution using NNT, the changes to the code are minimal. For example, in the NNT version of NCEP's Eta model [5], only 10% of the new code contained program lines that were added or modified (half of those were for I/O). Not only is it important that the number of changes introduced by NNT be minimal, but the resulting loop structures and computation lines must also be identical to the original versions, as shown in Section 3.

Several libraries have been proposed to parallelize weather prediction models. Among them, the GMD Grid Communication Library [4] and the Argonne National Laboratory's RSL [8] seem to be the most promising [2][9]. Other efforts to parallelize weather prediction models have been based on high-level libraries [1][7]. The GMD library is of more general scope than NNT, since it allows for non-regular domains. The cost of this generality is added complexity. RSL schedules individual columns of the 3D volume to each processor. This makes it simple to implement dynamic load balancing procedures but forces the programmer to change the original sequential code considerably. The advantages of NNT are the simplicity of its use, the treatment of I/O operations, and the possibility of overlapping computation, communication, and I/O operations.

In Section 2 we present a general description of NNT. Using examples from the Rapid Update Cycle (RUC) and Eta models, we describe in Section 3 a method for the parallelization of a sequential code using NNT. In Section 3 we also present performance results of different applications coded using NNT, and show how fully portable code that includes performance optimizations can be generated while still maintaining a simple programming interface. In Section 4 we compare the ease of programming of NNT and HPF the RUC model as an example. In Section 5 we study the execution

of redundant computes to reduce the amount of interprocessor communication, a technique that has proved very useful for weather forecast models. The optimizations possible with NNT that are described in this paper, must be included in HPF compilers in order to generate efficient code for weather forecast applications (this should hold for other applications as well). In Section 6 we study the cost of portability in NNT. We show the execution performance of a parallel Laplace solver coded using NNT which was run on the SGI Challenge multiprocessor. We also present the performance of the same model coded using the native programming tool of the machine (DOACROSS directive) to examine the penalty for using a local address space paradigm on multiprocessors with hardware support for global addressing. We show that under certain conditions the NNT code executes faster than the native SGI code.

2. GENERAL DESCRIPTION OF NNT

NNT was designed for the parallelization of regular grid-based weather prediction models. In this type of model, the volume of the forecast area is mapped into a regular three-dimensional grid, and a set of partial differential equations is solved for each point in the grid. The computations found in weather prediction models have characteristic stencils that define the spatial data dependencies for the computation of a particular grid point. For example, the equation

$$df(i,j) = \frac{1}{4} \cdot \left(f(i-1,j) + f(i+1,j) + f(i,j-1) + f(i,j+1) \right) - f(i,j)$$

has a 5 point stencil. We define the half-width of the stencil, in each dimension, as the maximum displacement in the index for that dimension. In the above equation, for example, the stencil half-width is 1 in the *i* and *j* dimension. The equation indicates that the computation of *df* in any grid point depends on the values of array *f* on the four neighboring points.

NNT provides a routine to map processes to data (domain decomposition). Each processor will have locally the set of data that corresponds to the region assigned to the process executing on the processor (*interior region*). NNT also manages a *halo region*, which is used to contain data that is

produced by neighboring processors and is needed for local computation. Halo regions of two rows and two columns are shown in Figure 1, for a two-dimensional decomposition. The data in the interior and halo regions can be input or output using the NNT I/O calls. Enough storage must be allocated in each array declaration to contain the interior and halo regions. NNT updates data in the halo regions using halo exchanges. The amount of data exchanged is given by the exchange *thickness*, which can be varied dynamically throughout the program execution. The maximum value of the exchange thickness is defined at the time of the data decomposition, and should be large enough to include the largest stencil related to a particular array.

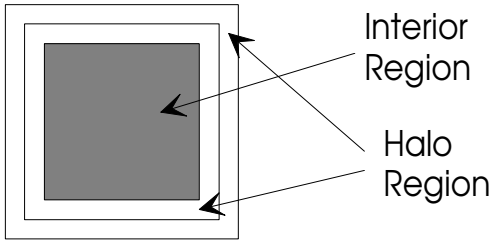

Figure 1. Interior and halo regions on a processor. In this example, the halo region consists of two rows and two columns in both dimensions.

The parallelization of a sequential code using NNT is based on data decomposition. Each process will execute the same program on different data. The first step in the parallelization process is to study the data dependencies in the code. These dependencies will define, for each array, the allowed decomposition types and the maximum thickness (stencil width) associated with it. The user is responsible for providing arrays large enough to hold the interior and halo regions. The specification of decomposition types and thicknesses can occur at run time. A data decomposition defines the regions assigned to each processor and also defines a virtual array of processors, creating neighboring relations between them. When the data decomposition routine is called, a decomposition structure is created and a structure handle is returned (a handle is a Fortran 77 integer). This handle

will be used as an argument to subsequent NNT calls to specify the decomposition that is the context to the specific NNT operation. Many arrays can be associated with the same decomposition.

To execute I/O or exchange operations, NNT uses structures that must be defined by the user (at run-time). These structures can be shared by many arrays, and allow for array aggregation so that multiple arrays can be input, output, or exchanged simultaneously. Handles to these structures are returned from routines that define the above operations. In Figure 2 we show the flow of creation and usage of NNT structures (accessed by handles). To define each operation the user must specify a decomposition handle and the number of arrays aggregated in the operation. To define an exchange operation a data type must be specified. The parameters for the I/O or exchange operations can also be modified at run time. This allows to change the selection of array operands and the exchange patterns and thickness for halo region exchanges.

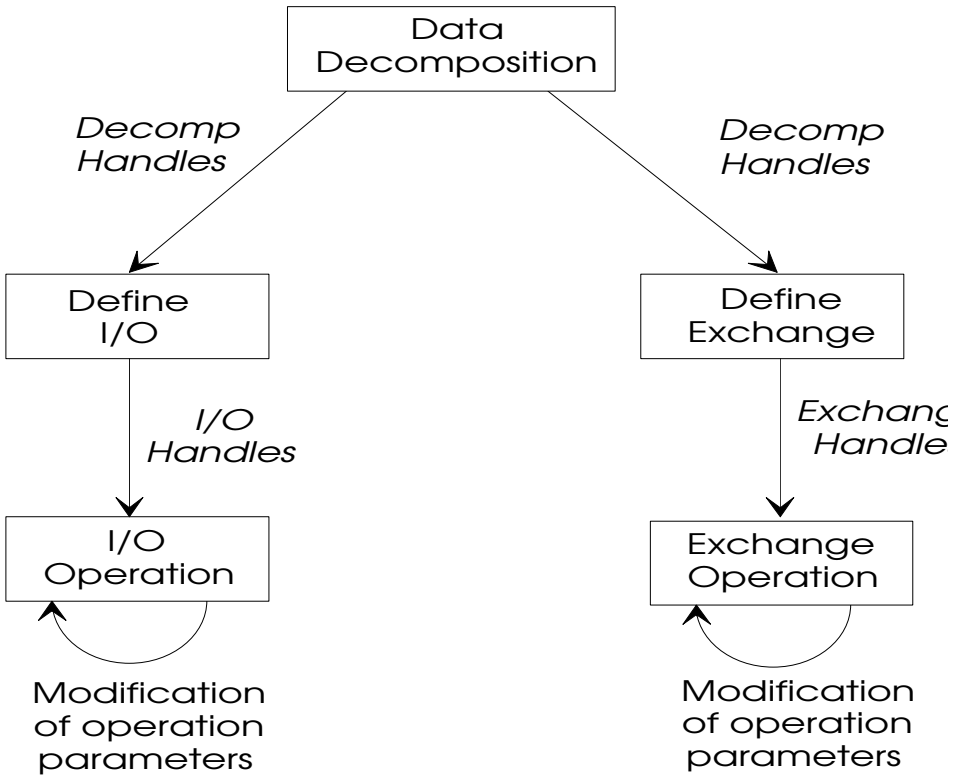

Figure 2. Dependence flow in the creation and usage on NNT handles

Once the decomposition, I/O, and exchange structures have been defined, the programmer inserts calls to NNT I/O and exchange functions where necessary. The remainder of the work is the loop index conversion. Since sequential codes are programmed in global address space, and NNT presents a local address view, the *start* and *end* parameters of loop structures must be changed to *start* and *end* values in the local space of each processor. NNT provides calls to do this translation.

3. PARALLELIZING A WEATHER FORECAST MODEL: EXAMPLES

To show how a weather prediction model is parallelized using NNT, we describe different aspects of this process using examples from the parallelization of the Eta and the RUC models [5][6]. A typical sequential program has I/O phases and computation phases. Figure 3 shows the typical flow of execution of an NNT program. In the NNT setup procedure, decomposition, I/O and exchange handles are created to define the NNT operations that will occur during execution. For large programs, it has been found useful to build a separate "NNT Setup" module that will call the NNT "define" routines. The handles can be passed to the rest of the program modules using common blocks. After the I/O and exchange handles are created, calls to NNT I/O and NNT exchange operations can be made. The computation part of a weather prediction program is primarily a set of loops. The *start* and *end* parameters of these loops can be modified using the NNT index translation routines (which can also be called from the NNT Setup module).

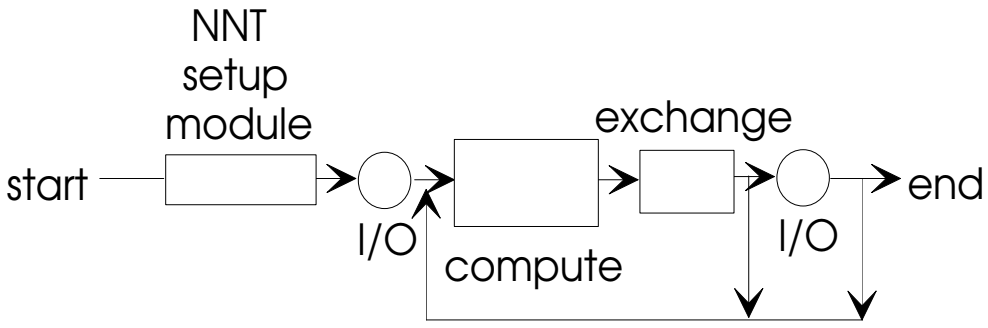

Figure 3. Typical flow of execution of an NNT program.

In the remainder of this section, we show examples of the usage of NNT. These examples were taken from the parallel versions of the Eta and RUC models. Details of the NNT calls are available in the NNT User's Guide [11].

3.1 Creation of NNT Handles

In the RUC model a common block in an *include* file was used to declare the handles (decomposition, I/O, and exchange handles) used by NNT and the loop index translation vectors used to translate loop *start* and *end* parameters. This include file was initialized by calls to NNT define functions. The following code defines a data decomposition structure (the necessary declarations are omitted):

```
1:        IDECOMPTYPE= NNT_DECOMP_1_2
2:        IBDYTYPE(1) = NNT_DECOMP_BDY_NONPER
3:        IBDYTYPE(2) = NNT_DECOMP_BDY_NONPER
4:        IBDYTYPE(3) = NNT_DECOMP_BDY_NONPER
5:        ITHICK(1) = I_THICKNESS
6:        ITHICK(2) = J_THICKNESS
7:        ITHICK(3) = 0
8:        IDATASIZE(1) = mix
9:        IDATASIZE(2) = mjx
10:       IDATASIZE(3) = 1
11:       CALL NNT_decomp(IDECOMPTYPE,IBDYTYPE,IDATASIZE,ITHICK,IDECOMP_2D,ISTAT)
```

Line 1 sets the decomposition type to a two-dimensional decomposition in the first two dimensions. Lines 2-4 set boundary types to nonperiodic. With periodic boundary conditions, the processors in the boundaries of the global domain exchange data with the processors in the opposite boundary. In

nonperiodic boundaries, processors in the boundary of the global domain do not exchange data outside of the boundary. Lines 5-7 set the boundary thickness, lines 8-10 set the size of the global array, and line 11 calls the NNT function that returns the decomposition handle `IDECOMP_2D`. Since Fortran 77 arrays are declared statically by the programmer, the information on the declared size is passed to NNT using the **`NNT_declared_size`** routine:

```
1:        ISTORAGESIZE(1) = mix_local
2:        ISTORAGESIZE(2) = mjx_local
3:        ISTORAGESIZE(3) = 1
4:        CALL NNT_declared_size(IDECOMP_2D,ISTORAGESIZE,ISTAT)
```

Lines 1-3 set the size used by the programmer in the array declaration, and line 4 calls the NNT function to set the declared size to the appropriate value. Since the size of the regions assigned to a processor decreases as the number of processors participating in the run increases, it is possible and generally beneficial to use the minimum declared array size for a given number of processors.(e.g. for NxM processors, $mix\_local = \frac{mix}{N} + 2 \cdot I\_THICKNESS$ and $mjx\_local = \frac{mjx}{M} + 2 \cdot J\_THICKNESS$.) This will optimize performance by improving the usage of the machine's memory hierarchy. To define a three-dimensional exchange the **`NNT_define_exch`** call is used:

```
1:        CALL NNT_define_exch(IDECOMP_3D,IEXCH_F_3D_5VAR,NNT_REAL,5,ISTAT)
```

The decomposition handle `IDECOMP_3D` is from a three-dimensional decomposition previously defined. The **`NNT_define_exch`** call creates an exchange handle (`IEXCH_F_3D_5VAR`) for an exchange of up to 5 real arrays. Assignment of model variables to an exchange handle is shown in Section 3.3.

The following are examples of the calculation of index translation vectors for translation of the *start* and *end* parameters in loop structures:

```
1:        CALL NNT_loops(IDECOMP_3D,1,I_START,I_END,1,NREGIONS,ISTAT)
2:        CALL NNT_loops(IDECOMP_3D,2,J_START,J_END,1,NREGIONS,ISTAT)
3:        CALL NNT_loops_nd(IDECOMP_3D,1,I_START_ND,I_END_ND,1,NREGIONS,ISTAT)
4:        ALL NNT_loops_nd(IDECOMP_3D,2,J_START_ND,J_END_ND,1, NREGIONS,ISTAT)
```

Lines 1 and 2 calculate the *start* and *end* parameters of the *I* (first) and *J* (second) dimensions of a loop structure that operates on arrays decomposed according to decomposition described by the handle `IDECOMP_3D`. The resulting vectors (`I_START`, `I_END`, `J_START`, `J_END`) will provide *start* and *end* indices to loop over the interior data of the processors. Lines 3 and 4 call **`NNT_loops_nd`**, which provides index translation vectors to loop over the data in the interior and halo regions. NNT also provides calls to get index translation vectors to loop over the interior data and a limited area of the halo region. The ability to perform computation in all or a portion of the halo region has proven very useful in optimizing parallel codes, as will be shown in Section 5. The computations in a processor's halo region are redundant, since those computations are also executed in a neighbor's interior region. If the data dependencies allow it, redundant computations can be traded for communication, since data in the halo region can be computed by distinct processors without need for communication. We show the use of the loop index translation vectors in Section 3.4. (The parameters 1 and `NREGIONS` are used for compatibility with future versions of NNT.) The following is the definition of an I/O operation taken from Eta:

```
1:        call NNT_make_io_handle(IO_2D_OUTUV,IDECOMP_2D,2,ISTAT)
```

This call creates an I/O handle (`IO_2D_OUTUV`) that defines the input and output operations of any array decomposed with a two-dimensional decomposition identified by the handle `IDECOMP_2D`. A total of 2 arrays can be read or written simultaneously per operation per process using the handle `IO_2D_OUTUV`.

3.2 I/O Operations

NNT provides a full set of I/O operations [14]. These include ASCII and binary I/O, and decomposed and nondecomposed array I/O. We only describe input and output operations on decomposed arrays. To execute an I/O operation, the arrays that will be the operands to the operation have to be assigned to the I/O handle. The following are examples taken from Eta:

```
1:   CALL NNT_assign_io_var(IO_2D_OUTUV,EGRID1_2d,1,ETAFLOATTYPE,ISTAT)
2:   CALL NNT_assign_io_var(IO_2D_OUTUV,EGRID2_2d,2,ETAFLOATTYPE,ISTAT)
```

Lines 1 and 2 assign arrays `EGRID1_2d` and `EGRID2_2d` as the first and second arrays to be operands of the I/O operation. Lines 1 and 2 are only necessary once, unless other arrays use the same I/O handle, and the array assignment of those lines is overwritten. I/O is performed using the following calls:

```
1:        CALL NNT_write_multi(IFIL_OUT,IO_2D_OUTUV,ISTAT)
2:        CALL NNT_flush_all(ISTAT)
```

Line 1 calls the NNT write routine, which writes the arrays into the file `IFIL_OUT`. The read routine has the same syntax. Line 2 is an optional call to the NNT flush routine. NNT I/O write operations buffer the data in separate nodes (*write buffers*) that serve as an interface to the storage system (a future version of NNT will also provide I/O buffering for read operations). This buffered data can be flushed at the time of the **`NNT_flush_all`**call, or automatically by the system when the buffers are full. The call to **`NNT_flush_all`**(that returns immediately) avoids having to wait for an automatic flush. The NNT write flush operation was used on the Eta and RUC models to overlap disk write time with processor activity. The number of write buffers active in a particular run is specified at run time, using the following syntax:

nntex NP executable NWB write_buffs

where NP and NWB are the number of processors and number of write I/O buffer processors for the run, respectively. I/O buffers have proven to reduce considerably the impact of I/O operations on program execution time. For example, in the RUC model with 40 km resolution over the continental US, the execution time on 144 processors of the Intel Paragon was reduced by an order of magnitude by adding 9 I/O buffer processors [13]. Experiments show that if there is enough processor activity to overlap with the write operations to disk, NNT write operations can completely hide the output time, as is done in the Eta and RUC model,.

3.3 Exchange Operations

Once an exchange operation has been defined, the arrays to be exchanged have to be assigned to the exchange structure and the exchange pattern must be set. The following is an example taken from the parallel version of the RUC model:

```
1:        CALL NNT_assign_var(IEXCH_F_3D_5VAR,1,ua,NNT_DO_ALL, ISTAT)
2:        CALL NNT_assign_var(IEXCH_F_3D_5VAR,2,va,NNT_DO_ALL,ISTAT)
3:        CALL NNT_assign_var(IEXCH_F_3D_5VAR,3,ta,NNT_DO_ALL,ISTAT)
4:        CALL NNT_assign_var(IEXCH_F_3D_5VAR,4,dfpa,NNT_DO_ALL,ISTAT)
5:        CALL NNT_assign_var(IEXCH_F_3D_5VAR,5,qva,NNT_DO_ALL,ISTAT)
```

Lines 1-5 assign arrays `ua`, `va`, `ta`, `dfpa`, and `qva` as arrays to be exchanged. Typically, these lines are necessary only once in a program. The exchange pattern (`NNT_DO_ALL`) is set to exchange halo regions between all neighbors, including the corner neighbors (a total of 8 neighbors in a two-dimensional decomposition, for example). The exchange operation is executed with the following call:

```
1:        CALL NNT_exchange(IEXCH_F_3D_5VAR,ISTAT)
```

The NNT exchange has as parameter the exchange handle. NNT also provides subroutines to overlap exchange operations with computation: **`NNT_start_exch`** and **`NNT_end_exch`** **`NNT_start_exch`** returns immediately, effectively overlapping the exchange time with the computations that precede the call to **`NNT_end_exch`** The syntax for these calls is the same as for **`NNT_exchange`**

NNT overlapped exchanges were used to improve the scalability of the Well-Posed Topographic model running on a network of Sun Sparc2 workstations linked through Ethernet (Figure 4). The halo region exchange was overlapped with the computation of the interior region assigned to each processor. The results show that performance optimizations were possible without sacrificing a simple programming interface.

3.4 Array Index Translation

Once the calls that compute the loop index translation vectors are made, the resulting vectors can be used to substitute *start* and *end* parameters in loop structures. The index translation vectors define the three-dimensional local space on which each processor should execute the loop operations. The following examples are from the RUC forecast model:

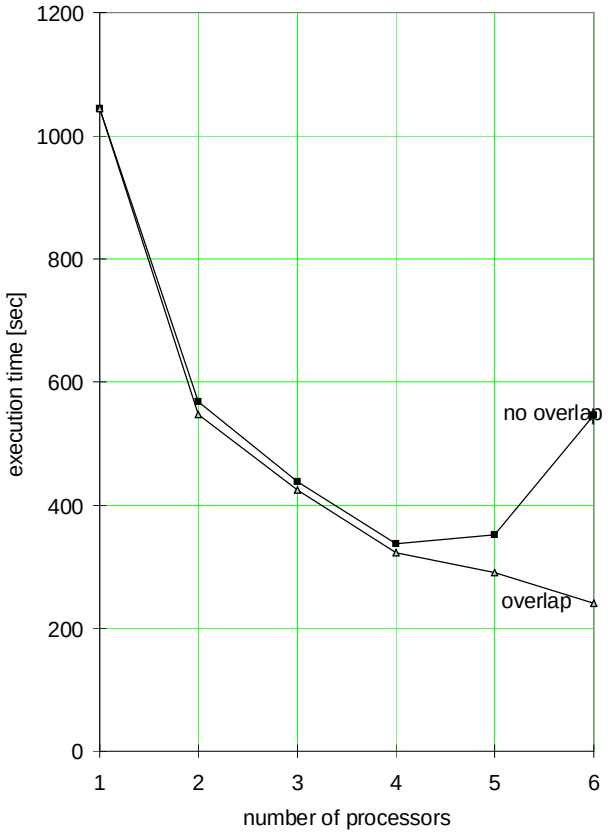

Figure 4. Performance of the Well-Posed Topographic model on a network of Sun Sparc2 workstations with Ethernet connection. Data size: 130x98x31, 32-bit precision.

Original sequential code:

```
1:        DO 122 j = 2,jlx
2:        DO 122 i = 2,ilx
3:        DO 122 k = 1,klx
4:        ka = max(k-1,1)
```

```
5:        qva(i,j,k) = qva(i,j,k)-.5*((sdot(i,j,k)+abs(sdot(i,j,k)))
   &           *(qvb(i,j,k)*qvb(i,j,k+1))-(sdot(i,j,k)-abs(sdot(i,j,k)))*
   &           (qvb(i,j,k)-qvb(i,j,ka)))/(p(i,j,ka )-p(i,j,k+1))
6:122     CONTINUE
```

Parallel NNT code:

```
1:        DO 122 j = J_START(2),J_END(jlx)
2:        DO 122 i = I_START(2),I_END(ilx)
3:        DO 122 k = 1,klx
4:        ka = max(k-1,1)
5:        qva(i,j,k) = qva(i,j,k)-.5*(sdot(i,j,k)+abs(sdot(i,j,k)))
     &    *(qvb(i,j,k)*qvb(i,j,k+1))-(sdot(i,j,k)-abs(sdot(i,j,k)))*
     &    (qvb(i,j,k)-qvb(i,j,ka)))/(p(i,j,ka )-p(i,j,k+1))
6: 122  CONTINUE
```

Since we use a two dimensional decomposition on *i* and *j*, the only changes to the code occur in lines 1 and 2, where the *start* and *end* parameters were replaced by the index translation vectors indexed with those start and end values. This greatly simplifies the translation process, which could be done by a preprocessor.

In summary, the parallelization of an application using NNT involves four steps: the creation of the NNT setup module, the substitution of loop index parameters, the insertion of exchange operations, and the substitution of I/O calls. The substitution of loop index parameters is trivial and could be done by a preprocessor, the insertion of exchange operations is guided by data dependence analysis necessary in any parallelization effort, and the substitution of I/O calls should be easy given the simple interface to the NNT I/O calls. The NNT setup module contains the basis for the parallelization of the application, and it defines the way data will be decomposed, exchanged, input, or output. This is where most of the calls to NNT will occur. It is our experience, however, that the structure of an NNT setup module can be reused between applications.

4. PROGRAMMABILITY: NNT AND HPF

The goal of this section is to compare NNT and HPF in programmability and performance. We will show that HPF provides a simpler user interface mainly because the programmer does not have to perform interloop dependence analysis. However, NNT allows for optimizations that cannot be expressed in HPF or carried out by current compilers. We study the performance impact of an optimization targeted to reduce communication costs, which has proven essential to run weather prediction models in real time. We hope that our discussions could motivate the development of compilers that perform such optimizations.

4.1 Domain Decomposition

The parallelization of any sequential code starts with study of the data dependencies. These dependencies define, for each domain decomposition, the data that must be moved between processors and the synchronization operations necessary to maintain correctness. The selection of a domain decomposition is based on the minimization of the effect of synchronization and data movement. Typically, this goal is achieved by increasing the ratio of computation to communication. In message passing architectures, the synchronization is embedded in the communication mechanism. Sometimes, program restructuring is used to modify data dependencies, which can result in reduced communication and synchronization costs.

Data dependence analysis must be done for each loop that will be parallelized. This analysis will define the set of domain decomposition strategies that allow parallelization, and should be carried out for both NNT and HPF programming. The domain decomposition is specified in HPF using the DISTRIBUTE directive (see [15] for a description of HPF). For example, to decompose the first two dimensions of a three dimensional array `U` on a two-dimensional plane of processors, the HPF syntax is:

```
1:        REAL U(NX, NY, NZ)
```

```
2:!HPF$   DISTRIBUTE U(BLOCK,BLOCK,*)
```

In NNT, the user, at run time, must specify the thickness of the halo regions (or maximum amount of data that will need to be communicated), and the periodicity of the boundaries (periodic or nonperiodic). If we set a vector `data_size` to `(NX, NY, NZ)`, each element of a vector `boundary_type` to `NNT_BDY_NONPERIODIC`, and each element of a vector `thickness` to the size of the halo region thickness in each dimension, then the syntax for specifying a domain decomposition in NNT is the following:

```
1:            CALL nnt_decomp(NNT_DECOMP_1_2, boundary_type,
         &    data_size, thickness, decomp_handle, status)
```

NNT does not force the user to bind the decomposition to a particular array. The binding is "indirect" since the decomposition handle (`decomp_handle`) is used in NNT operations like I/O and halo exchanges. In NNT, the user is responsible for declaring arrays that are large enough to hold the interior and halo regions assigned to each processor. In HPF the management of local address is invisible to the user (except maybe in the use of the NEW option on INDEPENDENT loops, as we will see later).

4.2 Loop Parallelization

Loop parallelization is done in HPF by means of the FORALL loop or the INDEPENDENT directive. In NNT a loop is "parallelized" using loop index translation vectors. These vectors are created using one of a group of NNT calls. Translation vectors are different across processors since they reflect the special treatment of physical boundaries and inequalities in work distribution. Translation vectors provide the *start* and *end* parameters of a loop so that each processor executes the body of the loop over its interior region (we will show later how NNT uses special translation vectors to reduce interprocessor communication). Consider the Loop 80, extracted from the RUC operational weather model:

LOOP 80, HYBCST.F, Sequential Version

```
1:        DO 80 K = 1,KKSIG
2:        DO 81 J = 1,JL
3:        DO 81 I = 1,IL
4:             HSCR5(I,J) = UB(I,J,K)
5:             HSCR6(I,J) = VB(I,J,K)
6:   81        HSCR1(I,J) = TB(I,J,K)
7:        CALL SMOLAR(HSCR1,HSCR2,HSCR5,HSCR6,MIX,MJX,IL,JL,DT,SCALE,BK)
8:        DO 82 J = 1,JL
9:        DO 82 I = 1,IL
10:82          TA(I,J,K) = HSCR1(I,J)
11:       IF (KI.GE.3) THEN
12:            DO 83 J = 1,JL
13:                 JA = 2*MAX0(J-1,  1)-(J-1)
14:                 JB = 2*MIN0(J+1,JL)-(J+1)
15:                 DO 83 I = 1,MIX
16:                      IA = 2*MAX0(I-1,  1)-(I-1)
17:                      IB = 2*MIN0(I+1,MIX)-(I+1)
18: 83                   HSCR3(I,J) = TB(I,J,K)
          &              -.25*(TB(IA ,J,K)+TB(IB ,J,K)
          &              +TB(I,JA ,K)+TB(I,JB ,K))
19:                 DO 84 I = 2,ILX
20:                 DO 84 J = 2,JLX
21: 84                   TA(I,J,K) = TA(I,J,K)-VELDFF*SCALE(I,J)*(4.*HSCR3(I,J)
          &              -HSCR3(I-1 ,J)-HSCR3(I+1 ,J)-HSCR3(I,J-1)-HSCR3(I,J+1))
          &              *DT/SCALP2(I,J)
22:            ENDIF
23: 80    CONTINUE
```

The HPF version of this loop, assuming that the corresponding DISTRIBUTE directives were used for the arrays inside the loops, is the following:

LOOP 80, HYBCST.F, HPF Version

```
1:        DO 80 K = 1,KKSIG
2:        FORALL (J=1:JL, I=1:IL)
3:             HSCR5(I,J) = UB(I,J,K)
4:             HSCR6(I,J) = VB(I,J,K)
5:             HSCR1(I,J) = TB(I,J,K)
6:        END FORALL
7:        CALL SMOLAR(HSCR1,HSCR2,HSCR5,HSCR6,MIX,MJX,IL,JL,DT,SCALE,BK)
8:        FORALL (J=1:JL, I=1:IL)
9:             TA(I,J,K) = HSCR1(I,J)
10:       END FORALL
11:       IF (KI.GE.3) THEN
12: ! HPF$    INDEPENDENT, NEW(JA,JB,I)
13:           DO 83 J = 1,JL
14:                JA = 2*MAX0(J-1,  1)-(J-1)
15:                JB = 2*MIN0(J+1,JL)-(J+1)
16: ! HPF$    INDEPENDENT, NEW(IA,IB)
17:                DO 83 I = 1,MIX
18:                     IA = 2*MAX0(I-1,  1)-(I-1)
19:                     IB = 2*MIN0(I+1,MIX)-(I+1)
20: 83                  HSCR3(I,J) = TB(I,J,K)-.25*(TB(IA ,J,K)+TB(IB ,J,K)
        &               +TB(I,JA ,K)+TB(I,JB ,K))
21:           FORALL (J=2:JLX, I=2:ILX)
22:                TA(I,J,K) = TA(I,J,K) - VELDFF*SCALE(I,J)*(4.*HSCR3(I,J)
        &          -HSCR3(I-1 ,J)-HSCR3(I+1 ,J)-HSCR3(I,J-1)-HSCR3(I,J+1))
```

```
      &          *DT/SCALP2(I,J)
23:           END FORALL
24:       ENDIF
25: 80  CONTINUE
```

The **INDEPENDENT** directives in Lines 12 and 16 include the **NEW** option since the variables JA, JB, I, IA, and IB should be made local to each processor, so that parallelization is not inhibited.

Assuming that the loop index translation vectors j_start, j_end, i_start, and i_end have been previously created using the **NNT_loops** routine, the parallelization of Loop 80 using NNT is the following :

LOOP 80, HYBCST.F, NNT Nonoptimized version

```
1:        DO 80 K = 1,KKSIG
2:        DO 81 J = J_START(1,0),J_END(JL,0)
3:        DO 81 I = I_START(1,0),I_END(IL,0)
4:             HSCR5(I,J) = UB(I,J,K)
5:             HSCR6(I,J) = VB(I,J,K)
6:81           HSCR1(I,J) = TB(I,J,K)
7:        CALL NNT_EXCHANGE(EXCH_HANDLE_1,STATUS)
8:        CALL SMOLAR(HSCR1,HSCR2,HSCR5,HSCR6,MIX_A,MJX_A,IL,JL,DT,SCALE,BK)
9:        DO 82 J = J_START(1,0),J_END(JL,0)
10:       DO 82 I = I_START(1,0),I_END(IL,0)
11:82          TA(I,J,K) = HSCR1(I,J)
12:       IF (KI.GE.3) THEN
13:            DO 83 J = J_START(1,0),J_END(JL,0)
14:                 JG = JGLOBAL(J)
15:                 JA = JTOLOCAL(2*MAX0(JG-1, 1)-(JG-1))
16:                 JB = JTOLOCAL(2*MIN0(JG+1,JL)-(JG+1))
17:                 DO 83 I = I_START(1,0),I_END(MIX,0)
```

```
18:                     IG = IGLOBAL(I)
19:                     IA = ITOLOCAL(2*MAX0(IG-1,  1)-(IG-1))
20:                     IB = ITOLOCAL(2*MIN0(IG+1,MIX)-(IG+1))
21: 83                  HSCR3(I,J) = TB(I,J,K)-.25*(TB(IA ,J,K)+TB(IB ,J,K)
        &               +TB(I,JA ,K)+TB(I,JB ,K))
22:           CALL NNT_EXCHANGE(EXCH_HANDLE_2,STATUS)
23:           DO 84 J = J_START(2,0),J_END(JLX,0)
24:           DO 84 I = I_START(2,0),I_END(ILX,0)
25:84              TA(I,J,K) = TA(I,J,K) - VELDFF*SCALE(I,J)*(4.*HSCR3(I,J)
      &            -HSCR3(I-1 ,J)-HSCR3(I+1 ,J)-HSCR3(I,J-1)-HSCR3(I,J+1))
      &            DT/SCALP2(I,J)
26:           ENDIF
27: 80    CONTINUE
```

An important distinction between HPF and NNT is that the user must perform interloop dependence analysis when programming using NNT. This analysis will define the need for halo regions and halo exchanges. The user must also define the size of the halo region. This process is handled by the compiler if HPF is used, and is probably the biggest advantage of HPF, since the process can be tedious and relatively complicated for large programs that call several subroutines. However, we will show that the user control over halo exchanges allows for optimizations that make NNT programs more efficient that their HPF counterpart. Lines 7 and 22 are calls to NNT exchange routines previously defined using the **`NNT_define_exch`**call. `EXCH_HANDLE_1` and `EXCH_HANDLE_2` are exchange handles created by the routine **`NNT_define_exch`**, which specify the data that must be exchanged. Line 7 exchanges HSCR5 and HSCR6 which need to be valid in the halo region for the execution of SMOLAR. Line 22 exchanges HSCR3, which needs to be valid in the halo region before the execution of Loop 84. In HPF the compiler is responsible for detecting the need and implementing the data communication. This process is well understood (see, for example, [16]). Lines 14-16 and

18-20 are necessary to compute local variables from global variables, a procedure that is done by the HPF compiler, guided by the NEW option.

Notice that the body of the loops in HPF and NNT are exactly like the ones of the original sequential code. In NNT, however, the values of variables that refer to the global domain (like J and I in lines 13, 14, 16, and 17 of the sequential code) must be obtained by mapping local values to the global domain (lines 14, 15, 16, 18, 19, and 20 of the NNT code). In HPF this is done by the compiler, but the user must specify variables that are local to processors, using the option NEW. The fundamental difference is that NNT provides the programmer with a local address space view, and HPF with a global address space view. The user must treat (NNT) or specify (HPF) exceptions to the "rules" suggested by the local address view (NNT) or the global address view (HPF).

In summary, programming with NNT is more complex than programming with HPF. To program with NNT the user must understand the local (across processors) treatment of the problem being solved. The user must:

- Perform loop and interloop data dependence analysis (only needed within loops in HPF).
- Define maximum "thickness" for halo exchanges.
- Provide enough storage to hold interior and halo regions, since dynamic memory allocation is not within the Fortran 77 standard.
- Map local values to global values (identify local variables in HPF).
- Create loop index translation vectors and replace *start* and *end* parameters in loops (insert FORALL or INDEPENDENT directive in HPF).

The changes that are made to the original code, however, are very similar in HPF and NNT. In both cases the loop statements are modified but most of the body of the loop remains the same.

5. TRADING REDUNDANT COMPUTATION FOR DECREASED COMMUNICATION IN NNT

Several optimization techniques have been used to optimize codes that are parallelized using NNT. Among them are the aggregation of arrays for exchange and I/O operations. In both cases the intent is to reduce the effects of the high latency associated with moving data between processors, and between processors and I/O devices. Another important optimization has been the overlapping of computation with communication. Again, this communication can be between processors (exchanges) or between processors and I/O devices (I/O operations). Overlapping of interprocessor communications has produced increases in performance on the order of 50% (as shown in Fig. 4). Array aggregation on I/O operations and the possibility of overlapping data reordering (which creates large contiguous blocks of data) and disk write time with other processor activity has resulted in performance gains of an order of magnitude [RUC][QNH].

The previous section showed that HPF is a parallelization approach that is simpler than high level libraries like NNT. However, for HPF to be successful, it must also perform well on multiprocessor systems. The optimizations described above must be carried out by HPF compilers. Array aggregation has been implemented by compilers [16]. Overlapping of data movement and computation might also be possible by a compiler, as is suggested by work that automatically generates prefetch calls for cache-based multiprocessors [17]. High performance I/O systems can be used by the HPF language if HPF defines appropriate syntax for parallel I/O operations. It is fundamental that the language/compiler be able to use systems that are capable of buffering and reordering data. In this section we present an optimization, possible with NNT, which is targeted to reduce communication by computing redundant operations inside a processor. We have measured performance improvements as high as 50%.

Consider the sequential version of Loop 80. In Loop 84 the computation of `TA(I,J)` needs data items `HSCR3(I-1,J)`, `HSCR3(I,J-1)`, and `HSCR3(I,J+1)`, among others with `(I,J)` indices. If data is decomposed in a two-dimensional decomposition, a particular processor will need data from the neighboring processor in the negative *I* direction, and the negative and positive *J* direction (processors on the physical boundary will behave differently). `HSCR3` is computed in Loop 83, and therefore it is exchanged in Line 22 of the nonoptimized NNT code. However, if each processor executes Loop 83 over its domain plus the inner row and column of the halo region (Fig. 1), the values of `HSCR3` that are

needed for the computation of Loop 84 on the interior region will reside in each processor. This assumes that the values of TB that are necessary to compute HSCR3 on Loop 84 are valid over the interior and halo regions (TB is not calculated on Loop 80, so let us assume that previous code guaranteed that TB is valid over the enlarged volume). NNT provides a simple way to guide the computation of a loop over sections of the halo region. In the program shown below (NNT Optimized) the *start* and *end* parameters for double loop 83 have a second index equal to 1. In the *start* parameters, a second index of 1 (*n*) indicates that each processor must execute the loop over the decomposed area, and also one (*n*) column(s) or row(s) in the negative direction. In the *end* parameters, a second index of 1 (*n*) indicates that each processor must execute the loop over the decomposed area, and also 1 (*n*) column(s) or row(s) in the positive direction. (Processors on a model boundary are treated appropriately by NNT.) The same optimization is used in Loop 81. The exchange used before the subroutine SMOLAR was avoided by computing HSCR5, HSCR6, and HSCR1, over the decomposed domain, plus two rows and two columns of the halo region. The computation over halo regions is said to be redundant because it is also carried out in the interior region of neighboring processors.

LOOP 80, HYBCST.F, NNT Optimized version

```
1:            DO 80 K = 1,KKSIG
2:            DO 81 J = J_START(1,2),J_END(JL,2)
3:            DO 81 I = I_START(1,2),I_END(IL,2)
4:                 HSCR5(I,J) = UB(I,J,K)
5:                 HSCR6(I,J) = VB(I,J,K)
6:81               HSCR1(I,J) = TB(I,J,K)
7:            CALL SMOLAR(HSCR1,HSCR2,HSCR5,HSCR6,MIX_A,MJX_A,IL,JL,DT,SCALE,BK)
8:            DO 82 J = J_START(1,0),J_END(JL,0)
9:            DO 82 I = I_START(1,0),I_END(IL,0)
10:82              TA(I,J,K) = HSCR1(I,J)
11:                IF (KI.GE.3) THEN
12:                     DO 83 J = J_START(1,1),J_END(JL,1)
```

```
13:                     JG = JGLOBAL(J)
14:                     JA = JTOLOCAL(2*MAX0(JG-1, 1)-(JG-1))
15:                     JB = JTOLOCAL(2*MIN0(JG+1,JL)-(JG+1))
16:                     DO 83 I = I_START(1,1),I_END(MIX,1)
17:                          IG = IGLOBAL(I)
18:                          IA = ITOLOCAL(2*MAX0(IG-1,  1)-(IG-1))
19:                          IB = ITOLOCAL(2*MIN0(IG+1,MIX)-(IG+1))
20:83                        HSCR3(I,J) = TB(I,J,K)-.25*(TB(IA ,J,K)+TB(IB ,J,K)
      &                      +TB(I,JA ,K)+TB(I,JB ,K))
21:           DO 84 J = J_START(2,0),J_END(JLX,0)
22:           DO 84 I = I_START(2,0),I_END(ILX,0)
23:84              TA(I,J,K) = TA(I,J,K) - VELDFF*SCALE(I,J)*(4.*HSCR3(I,J)
      &            -HSCR3(I-1 ,J)-HSCR3(I+1 ,J)-HSCR3(I,J-1)-HSCR3(I,J+1))
      &            *DT/SCALP2(I,J)
24:           ENDIF
25:80     CONTINUE
```

This optimization was used extensively in the RUC model to avoid exchanges inside vertical (*K*) loops, which included *I* and *J* loops, as in Loop 80. A total of 16 exchanges inside vertical loops and 3 exchanges inside the time-step loop were avoided by executing redundant computations. All the exchanges (three) were outside *k* loops, resulting in three exchanges per model time steps. Figure 5 shows the execution time of the RUC model on the Intel Paragon for optimized and nonoptimized versions of the NNT code. Times were measured using the UNIX "date" command before and after execution. Communication was not overlapped with computation. The write and read (I/O) time, combined, was always between 15 and 35 seconds. In the X axis, we show the number of compute processors plus the number of processors used as I/O buffers in each run. Above each bar, the percentage of execution time spent in halo exchanges is shown.

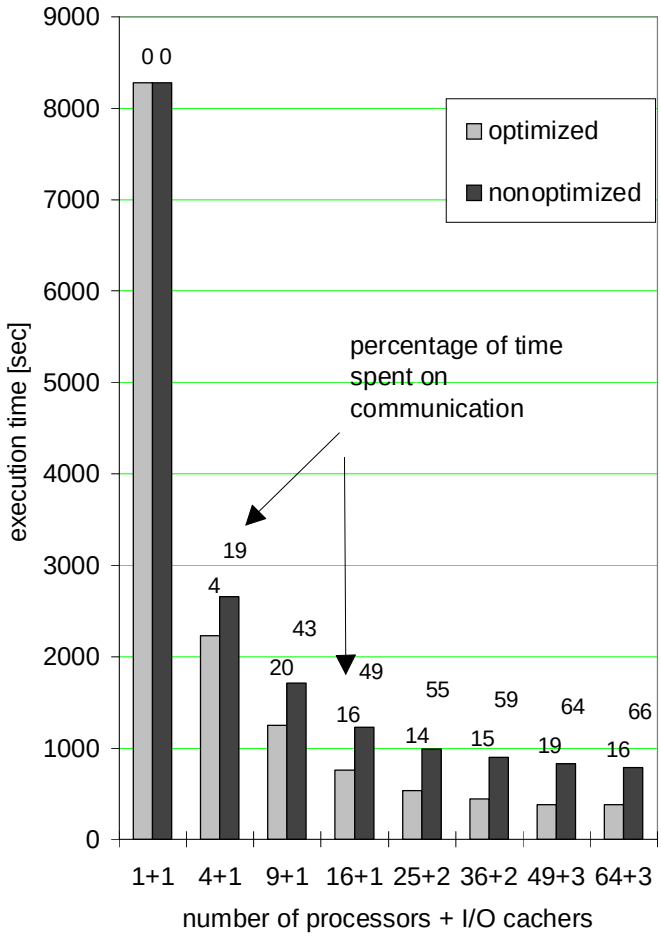

Figure 5. Total execution time of the 60 km RUC model, for a 12-hour forecast on the Intel Paragon, 32-bit precision.

For 64 processors and 3 I/O cachers, the optimized code ran approximately two times faster than the nonoptimized code. The percentage of time spent on halo exchanges was 16% for the optimized code and 66% for the nonoptimized code (remember, this is total execution time, including I/O and start-up). In Figure 6 we show the execution time of the RUC model running on an SGI Challenge multiprocessor (I/O cachers were not necessary). For two processors the optimized version of the code is slower than the nonoptimized. This occurs since data transmission is very fast and the cost of redundant computes outweighs the cost of additional halo exchanges. For 4 and 8 processors, on the

other hand, the contention for the interconnection bus (SGI Challenge architecture) considerably increases the effective transmission time, making the optimized version up to 1.7 times faster than the nonoptimized.

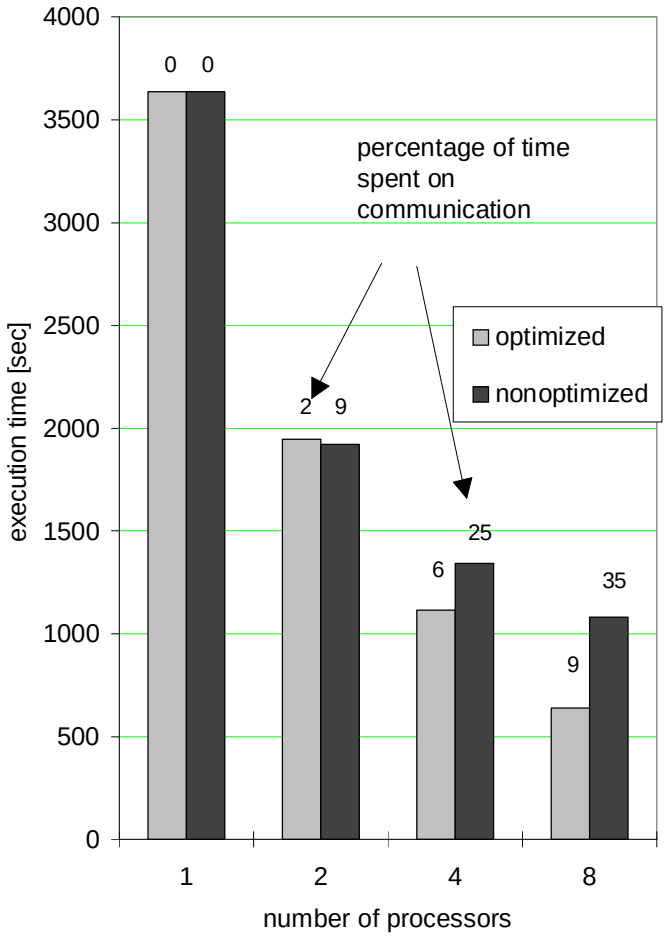

Figure 6. Total execution time of the 60 km RUC model, for a 12-h forecast on the SGI Challenge, 32-bit precision.

6. THE COST OF PORTABILITY IN NNT

One of the main reasons for designing NNT with a local address paradigm is that it allows portability between local and global address space architectures. The cost of portability is the inefficient use of global address space architectures if send and receive functions are called to move data between processors. We ported NNT to the SGI Challenge using System V Interprocess Communication (IPC) shared memory operations, which allow efficient movement of data during halo region exchanges. The data is gathered and scattered into a single shared buffer for communication, avoiding the high cost associated with the multilevel buffering used in message passing libraries [10].

We use a two-dimensional Laplace solver that uses a five-point stencil to probe the effectiveness of NNT on the SGI Challenge shared memory multiprocessor. We parallelized the Laplace solver using the DOACROSS directive and using NNT. The SGI DOACROSS directive only allows decomposition in one dimension. With NNT, however, the user can decompose in two dimensions. Figure 7 shows the execution time of the native SGI implementation (1D), the NNT implementations (1D and 2D), and the ideal execution time assuming perfect speed-up.

We can see in Fig. 7 that the execution times are very similar between the native SGI (DO ACROSS) and the 1D NNT implementation. The 2D NNT implementation, on the other hand, executes faster. (Remember that the difference between 1D and 2D versions of an NNT program is simply the selection of the decomposition type, which can be done at run time.) It is not possible to implement a two-dimensional decomposition using native SGI directives.

In general, global address space programs that decompose arrays on the dimension that is stored contiguously in memory will not use caches with large blocks effectively. Let us consider, for example, a two-dimensional problem that is decomposed in both dimensions. If the cache block contains b words, then the cache hit ratio will be $b\text{-}1/b$ during the access to the halo regions in the noncontiguous storage dimension. However, during the access to the halo regions in the contiguous dimension, the cache hit ratio will be $W\text{-}1/b$, where W is the stencil half-width in the contiguous dimension. NNT (and in general message passing) offers an "unexpected" benefit in programming shared memory multiprocessors with caches, since it improves cache utilization on multidimensional

decompositions. In message passing a processor copies data into a one-dimensional buffer before a send. The processor that later reads the buffer (receive operation) gets the full benefit of multiple-word fetch on a cache line load. Using message passing, a two-dimensional decomposition of the two-dimensional Laplace solver will be ideal, since it minimizes the ratio of communication to computation. Note that in a global address space multiprocessor with caches that have one-word blocks, a two dimensional decomposition would also minimize the number of coherence misses.

Error! Disk Full. Free some disk space and recalculate this field.

Figure 7. Execution time for a Laplace Solver on the SGI Challenge. The problem size was 512x512, and the algorithm used a five-point stencil, 32-bit precision.

Figure 7 shows that NNT is suitable for developing portable codes that run effectively on global address space architectures with small number of processors. We have not tested the global address port for large numbers of processors. The cost of using NNT on message passing architectures is negligible, related only to the overhead of two extra levels of subroutine calls on a halo region exchange.

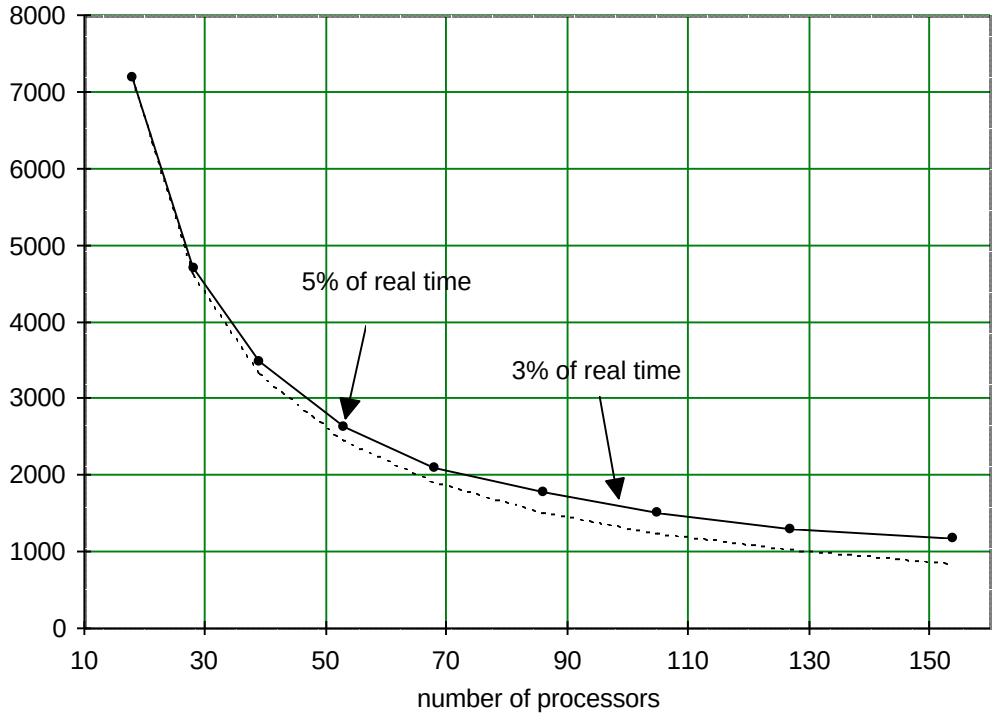

Figure 8. Execution time on the Intel Paragon for the RUC model, 40 km resolution, 12-hour forecast over the continental US. Dotted line indicates ideal scaling after 18 processors, 32-bit precision.

Figure 8 shows the execution time for a 12-hour forecast of the RUC model. The model resolution was 40 km over the continental US. The time includes the input of the initial model conditions (~30 MB) at the start of a run and the output of the atmosphere state (~18 MB) every hour of forecast, or 13 times per run. Operational weather forecast models must execute in 3% to 5% of the real time of the simulation (these points are indicated in the figure). The number of processors indicated in Fig. 8 includes the I/O buffers (see Section 3.2). Out of 153 processors, for example, 144 processors were compute processors and 9 were I/O buffer processors. For 144 processors, the addition of I/O buffers increased performance by an order of magnitude. The ideal execution time shown as a dotted line in the figure is obtained from perfect speed-up after 18 processors. The program was not run below 18 processors because paging to disk would have been necessary, resulting in very high execution times. The 40 km RUC model scaled well up to 153 processors. Early results indicate that higher resolution models will have better scaling.

7. CONCLUSIONS

NNT provides a set of subroutines that can be used for the parallelization of regular grid based weather prediction models. The Eta and RUC operational models have been parallelized using NNT. The definition of data decompositions, I/O operations, and exchanges can be localized in a single program module. The model code appearance is minimally affected, since only I/O operations and *start* and *end* parameters in loop structures need to be modified. Halo region exchanges need to be placed according to data dependence analysis. The code developed using NNT runs without source code change on any computing platform where NNT is ported. The data files also port between computers without changes.

HPF provides a portable mechanism for the parallelization of large codes that is simpler than high level libraries like NNT. HPF's biggest advantage is that the user need not study interloop dependencies and need not include explicit interprocessor communication calls. This is a fundamental benefit of global address space programming.

The execution of redundant computations to avoid data exchanges can reduce total execution time of operational weather models by approximately 50%, even on shared memory architectures. We have observed that the resulting communication time is in the order of 10-20% of execution time, and we believe that optimizations that overlap interprocessor communication with computation could completely hide this overhead. Compilers for languages like HPF should carry out these optimizations.

NNT provides an interface to a very fast parallel I/O system. Without such an I/O system, the execution times of weather models would increase by an order of magnitude or could not be programmed at all. It is essential to the weather prediction community that solutions like HPF supply an interface to fast I/O operations. We are currently starting efforts to combine the NNT and HPF advantages. A possible solution is to make a tool that uses information internal to HPF compilers to produce NNT code from HPF programs. Such tool will use the compiler's data dependence analysis to define halo regions and locate halo exchanges where appropriate.

For the types of problems investigated the performance of the codes that use NNT is comparable to the native implementation on machines with global address space supported in hardware. Moreover, NNT provides additional benefits on global address space multiprocessors with caches, since it allows for efficient cache use on multidimensional decompositions.

Acknowledgment. The authors would like to thank Clive Baillie for his suggestions on how to improve this paper. Funding for this project is being provided by the FAA/ARD-80, Aviation Weather Development Program Office. This work is supported by the Aviation Development Program.

8. REFERENCES

[1] A. Dickinson, P. Burton, E. Hibling, J. Parker, and R. Baxter. Implementation and initial results from a parallel version of the meteorological office atmosphere prediction model. *Proc. of the 6th Workshop on Use of Par. Proc. in Meteorology*. World Scientific, Reading, England, 1994.

[2] U.Gartel, W. Joppich, and A. Schuller. Parallelizing the ECMWF's weather forecast program: the 2D case. *Arbeitspapiere der GMD 740*, GMD, March,1993.

[3] L. Hart, T. Henderson, and B. Rodriguez. Evaluation of parallel processing technology for operational capability at the forecast systems laboratory. *Proc. of the 6th Workshop on Use of Par. Proc. in Meteorology*. World Scientific, Reading, England, 1994.

[4] R. Hempel, and H. Ritzdorf. The GMD communications library for grid-oriented problems. *Tech. Rep. 589*, GMD, Sankt Augustin, Germany, 1991.

[5] T. Henderson, C. Baillie, G. Carr, L. Hart, A. Marroquin, and B. Rodriguez. Parallelizing the Eta weather forecast model: initial results. *Proc. of High Performance Computing '94*. Society for Computer Simulation, La Joya, California, 1994.

[6] T. Henderson, C. Baillie, S. Benjamin, T. Black, R. Bleck, G. Carr, L. Hart, M. Govett, A. Marroquin, J. Middlecoff, and B. Rodriguez. Progress towards demonstrating operational capability of massively parallel processors at the Forecast Systems Laboratory. *Proc. of the 6th Workshop on Use of Par. Proc. in Meteorology*. World Scientific, Reading, England, 1994.

[7] T. Kauranne. The operational HIRLAM 2 model on parallel computers. *Proc. of the 6th Workshop on Use of Par. Proc. in Meteorology*. World Scientific, Reading, England, 1994.

[8] J. Michalakes. RSL: A parallel runtime system library for regular grid finite difference models using multiple nest. *Tech. Rep. ANL/MCS-TM-197,* Argonne Nat. Lab., Argonne, Illinois,1994.

[9] J. Michalakes and G. Grell. Parallel implementation, validation and performance of MM5. *Proc.*

of the 6th Work. on Use of Par. Proc. in Meteorology. World Scientific, Reading, England, 1994.

[10] B. Rodriguez, L. Hart, T. Henderson. Performance and portability in parallel computing: a weather forecast view. *Proc. of High Performance Computing in the Geosciences*. Kluwer Academic Publishers, Les Houches, France, 1993.

[11] B. Rodriguez, L. Hart, T. Henderson. NNT 1.0 user's guide. *Forecast Systems Laboratory Technical Memorandum*. in preparation.

[12] Message Passing Interface Forum. MPI: a message-passing interface standard. *International Journal of Supercomputing Applications*, Vol. 8, Number 3/4, (1994).

[13] T. Henderson, L. Hart, B. Rodriguez. High performance computing at FSL: demonstrating operational capability of massively parallel processors. *FSL Forum*. N. Fullerton (ed.), Forecast Systems Laboratory, March, 1995.

[14] L. Hart, T. Henderson, B. Rodriguez. An MPI based I/O support subsystem for a regular grid-based library. *Proc. MPI Developers Conference*, Notre Dame, Indiana, 1995.

[15] High Performance Fortran Forum, High Performance Fortran Journal of Development, *Scientific Programming*, Special Issue, Vol. 2, No. 1 and 2. (Spring and Summer 1993).

[16] S. Amarasinghe and M. Lam. Communication optimization and code generation for distributed memory machines. *Proc. of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*. Albuquerque, New Mexico, June 1993.

[17] T. Mowry, M. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching.

Proc. of the Fifth International Conference on Architectural Support for Programming Languages

and Operating Systems. Vol. 27, October 1992.