PORTING THE QUASI-NONHYDROSTATIC METEOROLOGICAL MODEL TO THE KENDALL SQUARE RESEARCH KSR1

C. F. Baillie and A.E. MacDonald, S. Sun

CU-CS-687-93

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

PORTING THE QUASI-NONHYDROSTATIC METEOROLOGICAL MODEL TO THE KENDALL SQUARE RESEARCH KSR1

CU-CS-687-93 November 1993

C.F. Baillie and A.E. MacDonald, S. Sun

Department of Computer Science
University of Colorado at Boulder
Campus Box 430
Boulder, Colorado 80309-0430 USA

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

Porting the Quasi-Nonhydrostatic Meteorological Model to the Kendall Square Research KSR1 *

C. F. BAILLIE
Computer Science
University of Colorado
Boulder CO 80309

and

A. E. MACDONALD, S. SUN
NOAA Forecast Systems Laboratory
Boulder CO 80303

ABSTRACT. The Forecast Systems Laboratory is currently developing and testing weather models on various parallel computers with a view toward ultimate production use. As part of that effort we have ported a "Quasi-Nonhydrostatic" Meteorological Model to the Kendall Square Research KSR1 parallel supercomputer. The porting was done in three stages: first the original sequential code was made efficient on one processor, then the code was parallelized using "tiling" directives, and lastly the parallel code was optimized. We describe this process in some detail and report the resulting performances.

1. Introduction

The "Quasi-Nonhydrostatic" (QNH) model is a non-hydrostatic, compressible meteorological model whose dynamics is based on the "approximate system" of Browning and Kreiss [1] plus the incorporation of topography as described in Browning and MacDonald [2]. QNH is a "full physics" model with packages for moist physics, radiation and turbulence. The dynamics part of QNH, known simply as the "well-posed" topographical model (WPT), was ported to the KSR1 by us previously [3]. We have built on the lessons learned there in doing this port of the complete QNH model. In addition, over the last year, we have substantially improved our understanding of the KSR1 architecture and its influence on performance. Therefore we shall begin with a detailed description of the KSR1 hardware as background so we can explain how one optimizes sequential codes on one processor. Next we shall briefly

* Talk given at Les Houches, 21-25 June 1993; to appear in "Proc. High Performance Computing in the Geosciences", ed. F.-X. Le Dimet (Kluwer Academic, Amsterdam, 1993)

outline the QNH model and code. Then the "tiling" method of parallelization will be described. Finally we give some details on how the resulting parallel code is optimized. Throughout we shall illustrate these steps with performance results for QNH.

2. The KSR1 and single processor code optimization

The Kendall Square Research KSR1 MPP (massively parallel processor) is the lastest shared-memory multiple-instruction, multiple-data (MIMD) computer [4]. It is built as a hierarchical machine, with a "fat-tree" type architecture, consisting of up to 34 rings of 32 processors. The memory is logically shared as there is one global address space, however it is physically distributed among the processors, each one having 32 MBytes. Most importantly the whole memory is treated as a cache, that is, whenever a processor gets data from another processor's memory or cache, a copy of this data is made in its own cache thereby significantly speeding up subsequent accesses to it. The rings are essentially hardware "search engines" resolving references to addresses that are not found in the processor's local cache by fetching the data automatically – hence all data movement is transparent to the user.

Unfortunately this wonderful feature for ease of programmability leads to some difficulties in obtaining high performance. The main problem is the actual hardware implementation of the cache. As is common in all mircoprocessors today, the KSR1 custom processor chip contains a cache, called the "subcache" in order to distinguish it from the 32 MByte local cache. This subcache is physically 2-way associative. This means that addresses separated by certain "magic numbers" get mapped into the same area of the subcache which can hold at most only two of them. Thus if several of these addresses are accessed by the program one after another, it gives rise to the problem known as "subcache thrashing" which seriously degrades performance. This pattern of address reference is in fact exactly what occurs in a Fortran program stepping through the second (or higher) dimension of an array (since Fortran arrays are stored in column-major order). The trivial fix for this problem is, of course, not to have arrays whose first dimension size leads to a magic number. This is most easily done by picking odd numbers which are not a power of two. Therefore in porting a code to the KSR1 the very first step is to change array declarations like $a(128, 128, 32)$ to $a(129, 129, 32)$ or equivalently $a(0 : 128, 0 : 128, 32)$. Coincidentally this is precisely what is done in codes designed for vector computers like the Cray but for a different reason, namely in order to avoid memory bank conflicts.

There is another single processor optimization which is crucial for certain codes like QNH on cache-based machines. This comes about because in caches, what is cached is not an individual variable of the program but a "cache line" which contains several variables. Therefore optimum use of the cache is obtained when the program accesses all of the variables on a cache line one after another before moving to the next cache line. This necessitates re-structuring conventionally written sequential scientific Fortran programs which typically update one variable or array element, then move onto another, then another, etc, as in the following example (taken from QNH) :

```
do k = 1,nz
 do j = 1,ny
  do i = 1,nx
   ffu = ffu + mu * (u(i+1,j,k) + u(i-1,j,k)
       + u(i,j+1,k) + u(i,j-1,k) - 4*u(i,j,k))
   ffv = ffv + mu * (v(i+1,j,k) + v(i-1,j,k)
       + v(i,j+1,k) + v(i,j-1,k) - 4*v(i,j,k))
   ffw = ffw + mu * (w(i+1,j,k) + w(i-1,j,k)
       + w(i,j+1,k) + w(i,j-1,k) - 4*w(i,j,k))
       + muz * (w(i,j,k+1) + w(i,j,k-1) - 2*w(i,j,k))
  end do
 end do
end do
```

We see that first a cluster of elements of u is modified, then similar clusters for v and w are modified. However, $v(i,j,k)$ and $w(i,j,k)$ are stored a long way from $u(i,j,k)$ in memory so there is not much data locality for the cache to take advantage of.

Instead the array elements should be "aggregated" or "coallesced" by declaring another array, say, $Q(3,0:nx,0:ny,nz)$ and copying $u(0:nx,0:ny,nz)$ into $Q(1,0:nx,0:ny,nz)$, $v(...)$ into $Q(2,...)$ and $w(...)$ into $Q(3,...)$. For readability one can also declare parameters $U=1, V=2, W=3$ so that $u(...)$ in the original code is now referred to as $Q(U,...)$. In order to do this aggregation simply without making mistakes some kind of program transformation tool is very helpful. We have been using the Sage++ tool from Gannon's group at Indiana [5]. To this we added the "user annotation" 'C$ann[Aggregate(u,v,w)]' which accomplishes the transformation just described, producing the following code :

```
do k = 1,nz
 do j = 1,ny
  do i = 1,nx
```

```
   ffu = ffu + mu * (Q(U,i+1,j,k) + Q(U,i-1,j,k)
       + Q(U,i,j+1,k) + Q(U,i,j-1,k) - 4*Q(U,i,j,k))
   ffv = ffv + mu * (Q(V,i+1,j,k) + Q(V,i-1,j,k)
       + Q(V,i,j+1,k) + Q(V,i,j-1,k) - 4*Q(V,i,j,k))
   ffw = ffw + mu * (Q(W,i+1,j,k) + Q(W,i-1,j,k)
       + Q(W,i,j+1,k) + Q(W,i,j-1,k) - 4*Q(W,i,j,k))
       + muz * (Q (W,i,j,k+1) + Q(W,i,j,k-1) - 2*Q(W,i,j,k))
  end do
 end do
end do
```

Now when the cluster of addresses around $Q(U,i,j,k)$ is fetched, similar addresses for $Q(V,i,j,k)$ and $Q(W,i,j,k)$ are pulled in on the same cache line (which on the KSR1 contains 16 64-bit variables) ready to be used next. In our codes this rearrangement of data yields up to a factor of two improvement in performance on the KSR1. We shall return to the actual performance numbers after outlining the QNH model and code.

3. The QNH Model and Code

The Quasi-Nonhydrostatic (QNH) meteorological model, like all regional forecast models, consists of two main parts: dynamics and physics. We already discussed the dynamics part of QNH in [3] so will not repeat it here; instead we just list the basic equations:

$$\frac{du}{dt} = -\frac{1}{\rho_o}\frac{\partial p}{\partial x} + fv + \nu\nabla^2 u \tag{1}$$

$$\frac{dv}{dt} = -\frac{1}{\rho_o}\frac{\partial p}{\partial y} - fu + \nu\nabla^2 v \tag{2}$$

$$\frac{dw}{dt} = \alpha(-\frac{1}{\rho_o}\frac{\partial p}{\partial z} + g\theta - \frac{gp}{\gamma P_o}) + \nu\nabla^2 w \tag{3}$$

$$\frac{dp}{dt} = -\gamma P_o(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}) + g\rho_o w + \nu\nabla^2 p \tag{4}$$

$$\frac{d\theta}{dt} = -\frac{1}{\theta}\frac{\partial\theta}{\partial z}w + \nu\nabla^2\theta \tag{5}$$

where

$$\alpha = (\frac{\Delta z}{\Delta x})^2 \tag{6}$$

is the "speed reduction" constant, which mathematically makes the equations symmetric and physically slows vertical sound waves down, so that the speed of sound in vertical and horizontal grid space is similar. u, v, w are the winds in the three dimensions, p is pressure, θ is temperature, ρ_0 is mean density in the vertical, ν is horizontal diffusion coefficient, f is Coriolis parameter, g is gravity constant, and γ is adiabatic exponent.

Recently, however, the dynamics part was modified to include well-posed boundaries which make the code somewhat harder to parallelize so we shall describe this new section of the dynamics. The theory of boundary conditions for the open boundary problem for symmetric hyperbolic systems was developed by Browning and Kreiss [6]. The so-called well-posed boundary conditions are based on the characteristics of the partial differential equations on each boundary. Consider a simple hyperbolic system describing sound wave propagation in a fluid moving at speed U_0. Let the speed of sound, $c >> U_0 > 0$. On the western boundary, two characteristics, $U_0 + c$ going from west to east, and $U_0 - c$ going from east from west, bring in the information from the exterior (either an outer coarse-mesh model or data) and from the interior, respectively. The combination of these two characteristics yields the final boundary values of u and p. If superscript c represents values obtained from the coarse mesh or data, n values from the inner numerical model, and $*$ the final boundary values, we have

$$u^* + p^*/c = u^c + p^c/c \quad ; \quad u^* - p^*/c = u^n - p^n/c \tag{7}$$

Thus the well-posed boundary values u^* and p^* are obtained by solving these equations.

The physics package in QNH currently includes moist physics, radiation and turbulence. The moist physics package is based on Schultz [7]. It contains six "moist variables": water vapor, cloud liquid, cloud ice, rain, snow and precipitated ice. The limited radiation package currently calculates heating associated with radiation. The Mellor-Yamada [8] Level 2.0 turbulence parameterization is used here for the turbulence in the boundary layer of a few kilometers thick.

The structure of the QNH code is as follows:

```
Loop over time

 Loop over space to calculate forcing functions
   from dynamics including advection, topography,
   horizontal diffusion, and physics
```

```
  Loop over space for dynamics time differencing
  Loops over horizontal boundaries to make them well-posed
  Loop over top vertical boundary to make it well-posed
  Loop over space for moist physics advection
  Loop over space for moist physics time differencing
  Loop over space for moist physics phase change
  Loop over space for radiation
  Loop over space for turbulence

End loop over time
```

All of the loops over space go over the whole horizontal plane $nx \times ny$ but they have different extents in the vertical. For dynamics the vertical loop goes all the way to the top of the atmosphere nz; for moist physics and radiation it goes approximately half-way; and for turbulence it is confined to a few layers near the ground. Hence when we come to parallelize the code we shall divide the three-dimensional space up into vertical columns, keeping all the various vertical loops local to each processor.

The original code was developed on a Sun Sparc 2 workstation which does not have any complicated caching so no care was taken in array size declaration (to avoid subcache thrashing) and no array aggregation was done (to improve use of the subcache), as explained above in Section 2. Therefore the original code ran badly on one processor of the KSR1, it also ran badly on the DEC 3000/400 workstation which is based on the DECchip 21064 or "Alpha" chip, as can be seen from the times labeled original in Table I. However, when the code was optimized by adding an extra row to the x and y dimensions of all the arrays and by aggregating the arrays for the primary variables, the speed on the KSR1 and on the DEC workstation increased by almost a factor of 2 – see the last column of Table I. We present the times for the DEC workstation to make the point that todays high performance workstations can benefit from simple code changes which improve data locality in their caches *in exactly the same way* as the KSR1 does. In fact the DEC Alpha chip is what Cray Research has chosen as processor in their first MPP product, the T3D. Thus the programming methodology we are developing for the KSR1 has wider applicability. We would also like to note that the clock speed of the KSR1 is only 20 MHz whereas that of the DEC 3000/400 is 133 MHz. Taking this into account we see that the KSR1 is actually performing better relatively, or putting it another way, speeding up the clock on the KSR1 will significantly improve its single processor performance relative to high end workstations.

TABLE I
Times in seconds for a short run of the original and the optimized versions of QNH with only dynamics and well-posed boundaries using a 32x32x21 grid on three different processors; the KSR1 is for a single processor.

Processor	original	optimized
Sun Sparc 2	31.1	30.4
KSR1	28.6	15.7
DEC 3000/400	7.9	4.5

We instrumented the code (again using the Sage++ tool) in order to discover that 15.7 seconds on the KSR1 corresponds to 7.7 Mflops. Here we have timed only the dynamics and well-posed boundaries to verify that the performance is about the same as what we obtained for the WPT code which is essentially just the dynamics part of QNH. In fact, for WPT [3] after array aggregation we obtained 7.8 Mflops. We actually went further with WPT and rearranged the code to bring updates to the same variables closer together thereby improving the performance to 8.6 Mflops. We could do the same here for QNH, but we elected to investigate parallel performance first.

4. Parallel Implementation on the KSR1

KSR provides several compiler directives in Fortran to aid parallelization of programs, the main one being "tiling" [9]. Essentially, tiling means splitting up a series of nested loops into blocks, each one of which can be executed independently of the others. In order to do this, all the user need do is add a few compiler directives. In our previous work on WPT [3] we explained tiling in some detail and tested various strategies to discover the best for our code. We shall not repeat all that here, suffice it to say that the conclusions for WPT apply to QNH. Namely that it is most efficient to tile the horizontal loops over nx and ny, with the vertical loop over nz in-processor. This horizontal tiling should be done using the so-called "slice" strategy which makes one tile per processor, and these tiles should be rectangular – long in the first dimension and thin in the second dimension (again due to Fortran array storage order). In fact highest performance is obtained when the tiles are $nx \times (ny/N)$, where N is the number of processors.

Of course in order to tile our loops over nx and ny we require a loop nest containing at least these two loops (and possibly also a loop over nz). Therefore we immediately have a problem with the well-posed horizontal boundary loops which go over only nx or ny but not both. The code for the ‘Loop over space for dynamics time differencing’, the ‘Loops over horizontal boundaries to make them well-posed’ and the ‘Loop over top vertical boundary to make it well-posed’ looks as follows :

```
do  k = 1,nz

TILE(i,j)
   do  j = 1,ny
     do  i = 1,nx
       Dynamics time differencing
     enddo
   enddo
   do  j = 1,ny
     Well-posed horizontal i = 1 bdy         ! west
   enddo
   do  j = 1,ny
     Well-posed horizontal i = nx bdy        ! east
   enddo
   do  i = 2,nx-1
     Well-posed horizontal j = 1 bdy         ! south
   enddo
   do  i = 2,nx-1
     Well-posed horizontal j = ny bdy        ! north
   enddo

enddo

TILE(i,j)
   do  j = 2,ny-1
     do  i = 2,nx-1
       Well-posed vertical k = nz bdy        ! top
     enddo
   enddo
```

We have indicated where the KSR tile directives would go with $TILE(i,j)$. Notice that the range of the second tile statement is two less than that of the first (2 to $ny-1$ and 2 to $nx-1$ rather than 1 to ny and 1 to nx). This could lead to unnecessary data movement between processors whose tiles change size unless both loops were included in an $AFFINITYREGION$ statement. This statement tells the compiler to keep the data fixed in the processors at the expense of having different tile sizes. Another way around this is to change the second set of loops to go over the whole $nx \times ny$ plane with an if statement inside to skip the edges; this is what we shall use. The reader may be wondering why we do not simply have one dimensional tile statements for the well-posed horizontal loops (i.e. $TILE(i)$ or $TILE(j)$). The problem with this is that it would cause a great deal of data to be moved around between processors since the two dimensional loops require rectangular tiles whereas the one dimensional loops require strips. Moreover for the i-loop these strips would be in one direction and for the j-loop in the other direction so effectively a data transpose would be happening between the i-loops and j-loops, in addition to the data movement between these loops and the two dimensional loops. The best solution is to obtain two-dimensional loops nested appropriately for tiling by rearranging the code as follows :

```
do  k = 1,nz

TILE(i,j)
   do  j = 1,ny
     do  i = 1,nx
       Dynamics time differencing
     enddo
   enddo

TILE(i,j)
   do  j = 1,ny
     do  i = 1,nx
       if (i .eq. 1) then
         Well-posed horizontal i = 1 bdy          ! west
       endif
       if (i .eq. nx) then
         Well-posed horizontal i = nx bdy         ! east
       endif
       if (i .ne. 1 .and. i .ne. nx) then
         if (j .eq. 1) then
           Well-posed horizontal j = 1 bdy        ! south
         endif
         if (j .eq. ny) then
           Well-posed horizontal j = ny bdy       ! north
         endif
         if (j .ne. 1 .and. j .ne. ny) then
           if (k .eq. nz) then
             Well-posed vertical k = nz bdy       ! top
           endif
         endif
       endif
     enddo
   enddo

enddo
```

All we have done is change the one dimensional loops into two dimensional loops plus if statements to select only the one dimensional parts we wanted in the first place. However this minor change is *essential* in order to be able to tile the loops with no data movement and achieve decent parallel performance. This is obvious from the times for the dynamics and well-posed boundaries part of the code listed in Table II. On, for example, 8 processors the two dimensional loop version obtains a speedup of 5.0 whereas the one dimensional loop version manages only 2.6. Notice that in addition to combining all four well-posed horizontal loops we also combined the well-posed vertical loop with them. This saves a tile statement with its associated overhead. We could go further and eliminate one more tile statement by combining our new loop over all the well-posed boundaries with the previous loop over the 'dynamics time differencing'. However this would change the order in which the boundaries are updated and therefore change the results of the code slightly. Nevertheless we tried doing this but as it only sped the code up by a few percent we decided it was not worth it after all. One last thing we tried was getting rid of the if statements to see how much time they were taking up – again it was only on the order of a percent. Incidentally, this rewrite of one-dimensional loops as two-dimensional loops will also help in porting the code to distributed memory message passing MPPs.

Now that we have the code rewritten entirely in the form of two dimensional loops we can go ahead and tile them all the same way to obtain good parallel performance. We show times for all seven loops, plus the total time, in Table III for a small run of the complete (dynamics and physics) QNH code on various numbers of KSR1 processors. In order to better see how efficiently the loops are being parallelized we calculate the speedups from these times, as shown in Table IV. It is clear that despite the overall speedups being fairly good (this is after all a rather small grid so there is not much parallelism left by the time we reach 16 processors) a large fraction of the loops are not yielding satisfactory speedups in themselves. Fortunately these

TABLE II
Times in seconds for a short run of the one dimensional loop (1D) and two dimensional loop (2D) versions of QNH with only dynamics and well-posed boundaries using a 32x32x21 grid on different numbers of processors of the KSR1.

Processors	1D	2D
1	15.6	15.7
2	11.8	8.4
4	7.4	4.4
8	5.0	2.6
16	4.1	1.5

TABLE III
Times in seconds for a short run of the complete QNH code using a 32x32x21 grid on different numbers of processors of the KSR1.

Loop	1 proc	2 procs	4 procs	8 procs	16 procs
Dynamics	12.7960	6.6405	3.4505	1.9175	1.0738
Dynamics time diff	3.1174	1.8070	1.1498	0.7429	0.4272
Moist physics adv	0.2422	0.1572	0.1146	0.0578	0.0466
Moist time diff	0.0589	0.0394	0.0202	0.0134	0.0116
Moist phase change	1.8627	0.9997	0.5149	0.2571	0.1370
Radiation	0.0192	0.0165	0.0099	0.0059	0.0045
Turbulence	0.0374	0.0243	0.0154	0.0073	0.0074
Total	18.15	9.73	5.33	3.06	1.77

loops consume only about 2% of the sequential run time. Most of the time (70%) is spent in the 'Dynamics' loop which shows the second best speedup. Next comes the 'Dynamics time difference' loop (which includes the two-dimensionalized well-posed boundary loop as described above) with 17% of the run time showing third best speedup. And last with 10% is the 'Moist physics phase change' loop which surprisingly shows the best speedup. We say surprisingly because this loop contains a great many if statements to implement all the possible phase changes between the six forms of moisture present. Therefore this part of code has the potential for the most load imbalance since areas of the sky with clouds will require more computing than blue sky. In tests with cloud covering one third of the sky this load imbalance was found to be less than 10% of the time spent in this loop.

Despite the remaining loops using such a small fraction of the run time we will discuss each of them in turn to explain why they show such poor speedups and indicate how to improve them. Firstly, the 'Radiation' loop

TABLE IV
Speedups from times in Table III.

Loop	1 proc	2 procs	4 procs	8 procs	16 procs
Dynamics	1.00	1.93	3.71	6.67	11.92
Dynamics time diff	1.00	1.73	2.71	4.20	7.30
Moist physics adv	1.00	1.54	2.11	4.19	5.20
Moist time diff	1.00	1.49	2.92	4.40	5.08
Moist phase change	1.00	1.86	3.62	7.25	13.60
Radiation	1.00	1.16	1.94	3.25	4.27
Turbulence	1.00	1.54	2.43	5.12	5.05
Total	1.00	1.87	3.41	5.93	10.25

TABLE V
Speedups for a short run of the complete QNH code using a 32x32x41 grid on different numbers of processors of the KSR1.

Loop	1 proc	2 procs	4 procs	8 procs	16 procs
Dynamics	1.00	2.24	4.50	8.45	14.50
Dynamics time diff	1.00	1.86	3.46	5.51	8.48
Moist physics adv	1.00	2.21	4.56	6.26	10.53
Moist time diff	1.00	1.95	3.90	7.62	9.05
Moist phase change	1.00	2.06	4.16	8.24	16.07
Radiation	1.00	1.76	3.75	5.28	7.24
Turbulence	1.00	1.47	3.88	7.30	11.22
Total	1.00	2.07	3.90	6.36	12.54

contains only a couple of statements so the overhead in tiling this loop is a large fraction of its run time. Moreover this overhead increases as the number of processors increases thereby limiting the speedup. The solution to this problem is to combine this loop with other loops in order to reduce the number of tile statements. Similarly the 'Moist physics advection' and 'Moist physics time difference' loops contain only a handful of statements, each one updating a different variable, so not enough work and little data reuse limits their speedup. In addition the vertical extent for these loops is less than half-way to the top of the atmosphere and this is the dimension which is in-processor so again there is not so much work to do for each tile creation and destruction. Lastly, the 'Turbulence' loop has two problems: it is the only loop in the code which has the vertical loop inside the horizontal loops, and this vertical loop runs over only two layers. Having the vertical loop inside the horizontal loops means that large strides are taken in the variables

TABLE VI
Times in seconds for a short run of the complete QNH using various sizes of grids on different numbers of processors of the KSR1.

Processors	32x32x21	64x64x21	128x128x21
1	18.15	100.3	706.9
2	9.73	47.3	319.9
4	5.33	23.4	133.9
8	3.06	12.4	60.2
16	1.77	6.9	29.3
32	-	4.8	18.8
64	-	-	12.1

TABLE VII
Speedups from times in Table VI.

Processors	32x32x21	64x64x21	128x128x21
1	1.00	1.0	1.0
2	1.87	2.1	2.2
4	3.41	4.3	5.3
8	5.93	8.1	11.7
16	10.25	14.5	24.1
32	-	20.9	37.6
64	-	-	58.4

(since the vertical index is the last one) which is bad for the subcache. And again with only two iterations of the vertical loop there is not enough work to ameliorate the overheads. To verify that the speedups would be better if there were more work in the loops we ran a grid of 32x32x41 with four layers for the turbulence. With this Table IV becomes Table V with much better speedups. In fact the two loops which exhibited the best speedups with 21 points in vertical now display superlinear speedups with 41 points. The 'Moist physics advection' loop shows the most dramatic improvement. The 'Radiation' is still the worst but is by far the smallest part of the code. Thus for maximum parallel performance it is important to have as many points in the vertical as possible. This corresponds physically to a high resolution in the vertical which is precisely what models like QNH are aiming for.

Finally we investigate how well the complete QNH code using various sizes of grids scales on different numbers of KSR1 processors. In Tables VI and VII we list the total run times and the speedups, respectively, for 32x32x21, 64x64x21 and 128x128x21 grids on up to 64 processors of the

TABLE VIII
Total Mflops from times in Table VII.

Processors	32x32x21	64x64x21	128x128x21
1	7.0	5.2	3.0
2	13.3	10.9	6.6
4	23.8	22.4	15.9
8	42.7	42.1	35.1
16	70.7	75.4	72.3
32	-	108.7	112.8
64	-	-	175.2

TABLE IX
Mflops per processor from times in Table VIII.

Processors	32x32x21	64x64x21	128x128x21
1	7.0	5.2	3.0
2	6.7	5.5	3.3
4	6.0	5.6	4.0
8	5.3	5.3	4.4
16	4.4	4.7	4.5
32	-	3.4	3.5
64	-	-	2.7

KSR1. We also convert run times to Mflops, the standard unit of performance, in Table VIII. With our instrumented version of the code we find that 18.2 seconds run time on one processor for the 32x32x21 grid corresponds to 7.0 Mflops. This is less than the 7.7 Mflops we obtained above for the dynamics only because the physics has more logical operations. Therefore a 64x64x21 grid, which requires four times as many flops, running in 100.3 seconds on one processor yields 5.2 Mflops. Similarly a 128x128x21 grid taking 706.9 seconds gives 3.0 Mflops. These performances for the larger sizes of grid are poor because the data does not fit in one processor's memory. However since the KSR1 is a shared memory machine the code still runs with the data being swapped to the memory of other processors. When the number of processors is such that the data just fits in their combined memory we obtain a peak in the Mflops per processor, which is shown in Table IX. For the three problem sizes this peak is for 1,4 and 16 processors respectively. We also obtain super-linear speedups (see Table VII) until we reach this number of processors. Unfortunately the Mflops per processor falls off significantly on large numbers of processors. This is true even when

the amount of work per processor is kept constant: comparing 32x32x21 on 1 processor, 64x64x21 on 4 processors and 128x128x21 on 16 processors in Table IX reveals that the Mflops per processor falls from 7.0 to 5.6 to 4.5. We are still not entirely sure why this happens (similar behavior was seen for WPT [3]) – but it is probably due to increased contention in the rings. Of course, the most important (some would say the only important) measure of performance, run time, decreases by about 60 on 64 processors. Therefore the KSR1 is very effective for running weather forecast codes in parallel.

The basic lesson we have learned in parallelizing code for the KSR1 is to minimize the number of tiled loops. Unfortunately this is just the opposite of what one does when vectorizing a code for the Cray. There the goal is to make every loop a vector loop over a few variables. However, as we discussed in Section 2, at least optimizing sequential codes for the two machines is similar.

5. Conclusions and Further Work

We have shown that it is fairly easy to port the QNH model to the KSR1 parallel supercomputer, obtaining reasonable sequential and parallel performance by making minor rearrangements of the data and code. We found a program transformation tool (such as Sage++) very useful in making these rearrangements. Our porting strategy is a three step process: first the original sequential code is made efficient on one processor, then the code is parallelized using "tiling" directives, and lastly the parallel code is optimized. The first step consists of three parts. Firstly an extra row and column is added to all the array dimensions to avoid subcache thrashing. Secondly arrays are aggregated (using solely directives to the program transformation tool) to help with data locality. Thirdly the code is rearranged to bring updates to the same variables closer together (we did not so this here but will do it before beginning production runs of QNH on the KSR1). The second step, namely adding tile statements, turns out to be the most straightforward. The last step is actually the most time consuming one as it involves rewriting one-dimensional boundary loops in two-dimensional form. As we pointed out the sequential optimization part of this strategy renders the sequential code efficient on other cache-based processor chips like the DECchip 21064 and on vector computers like the Cray; and the parallel optimization part helps with parallelization for other MPPs.

Our immediate future plans are to port the QNH model to the Intel Paragon distributed memory MPP. This involves a significant rewrite of the code but fortunately our colleagues have already done this for WPT (see [10] in these proceedings). Moreover they produced the so-called "nearest neighbor tool" which makes it much easier to do this rewrite by hiding a great deal of the message passing. Over the longer term, we are also involved in a project to port the National Meterological Center forecast model, called Eta [11], to MPPs. Based on our experience with WPT and QNH, we have decided to first port Eta to the KSR1 and then to the Intel Paragon, using our three part strategy.

Acknowledgements

CFB is supported by NSF Grand Challenge Applications Group Grant ASC-9217394 and by NASA HPCC Group Grant NAG5-2218. This work was funded in part by the NOAA Forecast Systems Laboratory.

References

[1] G. L. Browning and H.-O. Kreiss, 'Scaling and computation of smooth atmospheric motions', *Tellus* **38A** (1986) 295-313.

[2] G. L. Browning and A. E. MacDonald, 'Incorporating topography into the multiscale systems for the atmosphere and oceans', *Dynamics of Atmosphere and Oceans* **18** (1993) 119-149.

[3] C. F. Baillie and A. E. Macdonald, 'Porting the well-posed topographical meteorological model to the KSR parallel supercomputer', in: *Proc. Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology: Parallel Supercomputing in Atmospheric Science*, eds. G.-R. Hoffmann and T. Kauranne (World Scientific, London, 1993).

[4] 'KSR Parallel Programming' manual, *Kendall Square Research, 170 Tracer Lane, Waltham, MA 02154* (October 1991).

[5] D. Gannon, F. Bodin, S. Srinivas, N. Sundaresan, S. Narayana and J. Gotwals, 'Sage++, An Object Oriented Toolkit for Program Transformations', *Technical Report, Dept. of Computer Science, Indiana University* (1993).

[6] G. L. Browning and H.-O. Kreiss, 'Initialization of the shallow water equations with open boundaries by the bounded derivative method', *Tellus* **34** (1982) 334-351; and 'Scaling and computation of smooth atmospheric motions', *Tellus* **38A** (1986) 295-313.

[7] P. Schultz, 'Development and tests of a cloud physics parameterization for real-time aviation and public numerical weather forecasting', *NOAA Technical Memorandum ERL-FSL 6, U.S. GPO: 1993-774-025/89026* (1993).

[8] G. L. Mellor and T. Yamada, 'A hierarchy of turbulence closure models for planetary boundary layers', *J. Atmos. Sci.* **31** (1974) 1791-1806; and 'Development of a turbulence closure model for geophysical problems', *Rev. Geophys.* **20** (1974) 851-875.

[9] 'KSR Fortran Programming' manual, *Kendall Square Research, 170 Tracer Lane, Waltham, MA 02154* (October 1991).

[10] B. Rodriguez, L. Hart and T. Henderson, 'Performance and Portability in Parallel Computing: A Weather Forecast View', these proceedings.

[11] Z. I. Janjic, 'The step-mountain coordinate: Physical package', *Mon. Wea. Rev.* **118** (1990) 1429-1443.