Lawrence Livermore National Laboratory

CALTRANS

A Parallel, Deterministic, 3D Neutronics Code

Larry Carson, James Ferguson and John Rogers

Lawrence Livermore National Laboratory
P.O. Box 808, L-35, Livermore, CA 94550

Abstract

Our efforts to parallelize the deterministic solution of the neutron transport equation has culminated in a new neutronics code *CALTRANS*, which has full 3D capability. In this article, we describe the layout and algorithms of *CALTRANS* and present performance measurements of the code on a variety of platforms. Explicit implementation of the parallel algorithms of *CALTRANS* using both the function calls of the Parallel Virtual Machine software package (PVM 3.2) and the Meiko CS-2 tagged message passing library (based on the Intel NX/2 interface) are provided in appendices.

MASTER

ep

1 Introduction

With the arrival of the Meiko CS-2 machines, we have entered into a new era of computations in B Division. For the first time, it is our goal to write and/or convert large segments of major production codes in the Division from serial- to parallel-processing mode.

It is our intent to assist this effort with a contribution of our own: a new package of routines that solves the three-dimensional neutron transport equation on a set of parallel processors. We have dubbed this unclassified code *CALTRANS*. In our discussion below, we focus on the parallel algorithm used in *CALTRANS* and present timing studies of the code in action on a variety of serial and parallel platforms, including the Cray YMP, clusters of SGI Indigo and IBM RS6000 workstations, and our new Meiko CS-2 parallel computer. In particular, our work provides an example where it is *not* feasible, at least at the level of performance, to forgo special-purpose parallel computers such as the Meiko CS-2 for clusters of workstations working in parallel.

We also wish to aid the conversion effort by showing, in a concrete example, what it takes to write efficient parallel algorithms and how to analyze and predict the expected speed-up factors. This amounts to recognizing and dealing with the constraints of Amdahl's Law. Our discussion shall be as pedantic as possible; for example, the specific control structures that are needed to distribute work amongst processors are illustrated with explicit code fragments. Furthermore, in two appendices, we give all details on installing and running the code on the Meiko CS-2, as well as give the PVM interface routines that we wrote to run *CALTRANS* on workstation networks. Our purpose is to demonstrate that, at least for neutronics codes, the conversion to parallel processing is rather simple.

In the next section, we present the neutron transport equation and the physical and finite-element approximations that are made to cast it in a form amenable to numerical treatment. These approximations are applied to the transport equation to secure the actual finite-difference equations that are solved by *CALTRANS*. In section 3, we focus attention to the algorithm we use for solving this system of equations, and describe in section 4 the steps we take to parallelize it. The heuristic we shall employ is to identify the largest loops in the computationally intensive portion of our code, and to parallelize those.

In Section 5, we see how well we did by timing relevant portions of the code when applied to a number of fixed source and criticality problems, and running on several different platforms. The results are analyzed for scaling in the appropriate variables (number of zones, etc.), with particular attention paid to machine-dependent parameters such as the latency and bandwidth of interprocessor communication. This analysis will permit us to extrapolate to larger problems on a larger number of processors. As we shall see, multi-processing machines such as the Meiko CS-2, which minimize interprocessor communication times, produce superior performance.

We conclude in section 6 with a shopping list of improvements that we plan to bring to the code in the near future. In addition, we have provided two appendices for readers wishing to examine explicit code implementing Intel NX/2 message passing in PVM, as well as an example of initialization code and commands for running on the Meiko CS-2.

2 The Transport Equation

The object of our calculations is to determine the neutron phase space distribution $N(\mathbf{x}, \mathbf{p}, t)$ which describes the number of neutrons located in the incremental phase space volume $d^3\mathbf{x}d^3\mathbf{p}$ at the point $\mathbf{x}$ with momentum $\mathbf{p}$. We shall presume that the neutrons are nonrelativistic with kinetic energies E that are insignificant compared to the neutron rest mass energy mc^2 . In this case, it is more convenient to evaluate the neutron angular flux distribution $\psi(\mathbf{x}, E, \mathbf{\Omega}, t) = vN(\mathbf{x}, m\mathbf{v}, t)$, where the neutron velocity $\mathbf{v} = v\mathbf{\Omega}$ is expressed in terms of its magnitude v and direction $\mathbf{\Omega}$.

In the absence of interactions, the angular flux flowing in a given direction $\mathbf{\Omega}$ at a given energy E is unchanged in time from the point of view of an observer that moves with the neutrons. This free advection is expressed by the equation,

$$\frac{1}{v} \frac{D}{Dt} \psi(\mathbf{x}, E, \mathbf{\Omega}, t) = \left(\frac{1}{v} \frac{\partial}{\partial t} - \mathbf{\Omega} \cdot \nabla \right) \psi(\mathbf{x}, E, \mathbf{\Omega}, t) = 0.$$

Of course, neutrons do experience interactions, either by scattering off or being absorbed by nuclei in the background medium, or by causing nuclei to fission with a subsequent release of fission neutrons. In addition, radioactive sources may be present which feed neutrons into the system. The current version of *CALTRANS* does not model fission processes, so we shall ignore this source of neutrons. The other interactions modify the equation above to read,

$$\left(\frac{1}{v} \frac{\partial}{\partial t} + \mathbf{\Omega} \cdot \nabla + \sigma_T(\mathbf{x}, E, t) \right) \psi(\mathbf{x}, E, \mathbf{\Omega}, t) = S(\mathbf{x}, E, \mathbf{\Omega}, t), \quad (1)$$

where the source term on the right-hand-side of this equation is given by

$$S(\mathbf{x}, E, \mathbf{\Omega}, t) = Q(\mathbf{x}, E, \mathbf{\Omega}, t) + \int dE' \int \frac{d\mathbf{\Omega}'}{4\pi} \sigma_S(\mathbf{x}, E, E', \mathbf{\Omega} \cdot \mathbf{\Omega}', t) \psi(\mathbf{x}, E', \mathbf{\Omega}', t). \quad (2)$$

Here $Q(\mathbf{x}, E, \mathbf{\Omega}, t)$ is an external source term giving the number of neutrons emitted per unit volume per unit time in the energy interval $(E, E + dE)$ in the directions $(\mathbf{\Omega}, \mathbf{\Omega} + d\mathbf{\Omega})$ at the point $\mathbf{x}$. The quantities σ_T and σ_S are the *macroscopic* total and scattering cross sections, respectively, and possess units of an inverse length. The relation between these macroscopic cross sections and the time- and space-independent microscopic cross sections τ is given, for an homogenous material of molecular weight A and local mass density $\rho(\mathbf{x})$, by the expression,

$$\sigma(\mathbf{x}, t) = \frac{N_A \rho(\mathbf{x}, t) \tau}{A},$$

where N_A is Avogadro's number. For a heterogenous material consisting of N different molecular species, this formula generalizes to

$$\sigma(\mathbf{x}, t) = \sum_{n=1}^N \frac{N_A \rho^{(n)}(\mathbf{x}, t) \tau^{(n)}}{A^{(n)}}, \quad (3)$$

where $A^{(n)}$ is the molecular weight, $\rho^{(n)}$ the partial density and $\tau^{(n)}$ the associated microscopic cross section of each molecular species. The latter are simply constructed from the cross sections of every atom in the molecule, appropriately weighted by its stoichiometric factors.

As shown by Case and Zwiefel [1], the transport equation (1) has a unique solution in a volume $\mathcal{V}$ defined by a surface $\mathcal{S}$ whenever the initial flux $\psi(\mathbf{x}, E, \boldsymbol{\Omega}, t = 0)$ and external source $Q(\mathbf{x}, E, \boldsymbol{\Omega}, t)$ are specified everywhere in the volume $\mathcal{V}$, and the incoming flux is specified everywhere on the surface $\mathcal{S}$. In the current version of *CALTRANS*, we assume that this incoming flux is zero, so that if $\mathbf{n}$ is an outward normal of $\mathcal{S}$, then our boundary condition reads

$$\psi(\mathbf{x}, E, \boldsymbol{\Omega}, t) = 0 \text{ for } \mathbf{n} \cdot \boldsymbol{\Omega} < 0, \mathbf{x} \in \mathcal{S}. \quad (4)$$

While *CALTRANS* does not yet model fission processes, it is possible to mock up fission processes by treating the scattering term of the source (2) *as if* it were a fission source. For benchmarking purposes, we have found it convenient to perform criticality calculations using such pseudo-fission sources. In this case, we shall assume no external sources are present, and modify (2) to read

$$S(\mathbf{x}, E, \boldsymbol{\Omega}, t) = \frac{1}{\kappa} \int dE' \int \frac{d\Omega'}{4\pi} \sigma_S(\mathbf{x}, E, E', \boldsymbol{\Omega} \cdot \boldsymbol{\Omega}', t) \psi(\mathbf{x}, E', \boldsymbol{\Omega}', t). \quad (5)$$

For this source, the transport equation (1) is homogeneous and therefore is an eigenvalue equation in κ . Systems with $\kappa < 1$ are subcritical, while those with $\kappa > 1$ are supercritical.

Several physical assumptions are implicit in Eq. (1). The most important ones are:

- Fission sources and neutron-neutron scattering are neglected.
- The medium is isotropic, so that the total cross section σ_T is independent of the flux direction $\boldsymbol{\Omega}$, while the scattering cross section σ_S depends only on the scattering angle $\theta_S = \cos^{-1}(\boldsymbol{\Omega} \cdot \boldsymbol{\Omega}')$.
- The interactions are assumed to take place instantaneously, or at least quickly on the relevant time scales of the dynamical system. The latter depend on the distribution of neutron velocities together with the length scales of spatial variation in the medium. In particular, this neglects the contribution of delayed neutrons from slow fission processes, which are in any case ignored in this version of *CALTRANS*.

These assumptions are rather standard (cf. Lewis and Miller [2]). The current version of *CALTRANS* actually solves a more limited form of Eq. (1), obtained by imposing the following additional conditions:

- The external source and scattering cross section are isotropic.
- The macroscopic cross sections (e.g. the material densities $\rho^{(n)}$) and external source Q are time-independent.

As a consequence of the second condition, the angular flux ψ is also time-independent. Therefore, the (static) transport equation we actually solve is

$$(\mathbf{\Omega} \cdot \nabla + \sigma_T(\mathbf{x}, E)) \psi(\mathbf{x}, E, \mathbf{\Omega}) = S(\mathbf{x}, E), \quad (6)$$

with

$$S(\mathbf{x}, E) = \begin{cases} Q(\mathbf{x}, E) + \int dE' \sigma_s(\mathbf{x}, E, E') \phi(\mathbf{x}, E'), & \text{fixed source calculation,} \\ \frac{1}{\kappa} \int dE' \sigma_s(\mathbf{x}, E, E') \phi(\mathbf{x}, E'), & \text{criticality calculation.} \end{cases} \quad (7)$$

In this last equation, we have defined the scalar flux,

$$\phi(\mathbf{x}, E) = \int \frac{d\mathbf{\Omega}}{4\pi} \psi(\mathbf{x}, E, \mathbf{\Omega}). \quad (8)$$

At this stage, we must form a discrete version of Eqs. (6) and (7) that is amenable to numerical solution. For the directions $\mathbf{\Omega}$ we shall select a set of discrete angles, $\mathbf{\Omega}_m$, $m = 1, \dots, M$, uniformly distributed over the unit sphere, and only compute the angular fluxes ψ along these particular directions. Dependence of the flux on the energy variable E is truncated to a finite range (E_{min}, E_{max}) and is furthermore averaged over energy in a set of judiciously chosen *energy groups*. The latter are finite energy intervals (E_g, E_{g-1}), $g = 1, \dots, G$, with $E_{min} = E_G < E_{G-1} < \dots < E_1 < E_0 = E_{max}$. Hence, our flux variables are approximated by the set,

$$\psi_{gm}(\mathbf{x}) = \frac{1}{\Delta E_g} \int_{E_g}^{E_{g-1}} dE \psi(\mathbf{x}, E, \mathbf{\Omega}_m), \quad (9)$$

while expression (8) for the scalar flux reduces to the discrete sum,

$$\phi_g(\mathbf{x}) = \frac{1}{M} \sum_{m=1}^M \psi_{gm}(\mathbf{x}). \quad (10)$$

Averaging the transport equation (6) over energies is less straightforward; one usually approximates as follows:

$$\begin{aligned} \frac{1}{\Delta E_g} \int_{E_g}^{E_{g-1}} dE \sigma_T(\mathbf{x}, E) \psi(\mathbf{x}, E, \mathbf{\Omega}_m) &= \frac{\int_{E_g}^{E_{g-1}} dE \sigma_T(\mathbf{x}, E) \psi(\mathbf{x}, E, \mathbf{\Omega}_m)}{\int_{E_g}^{E_{g-1}} dE \psi(\mathbf{x}, E, \mathbf{\Omega}_m)} \psi_{gm}(\mathbf{x}) \\ &\simeq \frac{\int_{E_g}^{E_{g-1}} dE \sigma_T(\mathbf{x}, E) \phi(\mathbf{x}, E)}{\int_{E_g}^{E_{g-1}} dE \phi(\mathbf{x}, E)} \psi_{gm}(\mathbf{x}) \\ &\simeq \sigma_{T,g}(\mathbf{x}) \psi_{gm}(\mathbf{x}). \end{aligned}$$

The first approximation assumes that, up to overall scale factors, the energy-averaged ψ and $\sigma_T \psi$ distributions are angle-independent. The error introduced by the second approximation depends in detail on how the weighted cross section averages, defined by

$$\sigma_{T,g}(\mathbf{x}) = \int_{E_g}^{E_{g+1}} dE w_g(E) \sigma_T(\mathbf{x}, E),$$

are constructed. Ideally, the weight factors $w_g(E)$ should be computed according to the actual distribution of scalar fluxes, obtained by iteratively adjusting the input scalar fluxes to match those output by solving the equation. In practice, the weight factors are assigned fixed generic values depending on the problem one is trying to solve. Needless to say, it is difficult to gauge the magnitude of error introduced in this fashion; further research on the question is required.

If we perform the energy and angle discretizations just outlined, the transport equation (Eqs. (6-7)) becomes

$$\begin{aligned} (\boldsymbol{\Omega}_m \cdot \nabla + \sigma_{T,g}(\mathbf{x})) \psi_{gm}(\mathbf{x}) &= S_g(\mathbf{x}), \\ S_g(\mathbf{x}) &= \begin{cases} Q_g(\mathbf{x}) + \sum_{g'=1}^G \sigma_{S,gg'}(\mathbf{x}) \phi_{g'}(\mathbf{x}), & \text{fixed source calculation,} \\ \frac{1}{\kappa} \sum_{g'=1}^G \sigma_{S,gg'}(\mathbf{x}) \phi_{g'}(\mathbf{x}), & \text{criticality calculation,} \end{cases} \end{aligned} \quad (11)$$

where $g = 1, \dots, G$, $m = 1, \dots, M$, and ϕ_g is given by Eq. (10). The macroscopic total and scattering cross sections are related by formula (3) to their microscopic counterparts, $\tau_{T,g}^{(n)}$ and $\tau_{S,gg'}^{(n)}$, $n = 1, \dots, N$, for all N materials in the problem. By binning the energy dependence of the transport equation into energy groups, we have also changed the qualitative interpretation of $\sigma_{T,g}$, which now includes within-group scattering contributions in addition to absorption. However, these additional contributions introduced by the energy discretization are cancelled by equivalent terms in the scattering source $S_g(\mathbf{x})$. To distinguish from pure absorption, we shall call $\sigma_{T,g}$ the “removal” cross section.

Unfortunately, as a consequence of the angular discretization, the solutions of Eq. (11) often display preferential propagation along the discrete directions $\boldsymbol{\Omega}_m$, an unphysical effect known as the *ray effect* (see, for instance, Ref. [2]). The problem is particularly acute whenever localized sources are present, such as point sources or beams of neutrons. *CALTRANS* contains an option that may partially eliminate these ray effects in the case of a single, point, isotropic source, that is, when $Q_g(\mathbf{x})$ is given by the distribution,

$$Q_g(\mathbf{x}) = q_g \delta(\mathbf{x} - \mathbf{x}_0).$$

This describes a point source is located at $\mathbf{x}_0$ emitting q_g neutrons per second in each energy group g . The option uses a technique known as the *first-scatter method*, which separates the flux $\psi_{gm}(\mathbf{x})$ into two contributions,

$$\psi_{gm}(\mathbf{x}) = \psi_{gm}^0(\mathbf{x}) + \delta\psi_{gm}(\mathbf{x}), \quad (12)$$

where $\psi_{gm}^0(\mathbf{x})$ is the unscattered flux emanating directly from source, which may however be absorbed. It is a solution of Eq. (11) with the scattering term turned off ($\sigma_{S,gg'} \rightarrow 0$), and has the analytic form,

$$\psi_{gm}^0(\mathbf{x}) = \frac{q_g \exp(-\gamma_g(\mathbf{x}, \mathbf{x}_0))}{|\mathbf{x} - \mathbf{x}_0|^2} \delta\left(\frac{\mathbf{x} - \mathbf{x}_0}{|\mathbf{x} - \mathbf{x}_0|} - \boldsymbol{\Omega}_m\right), \quad (13)$$

where the *optical thickness*, in group g , between $\mathbf{x}$ and $\mathbf{x}_0$ is

$$\gamma_g(\mathbf{x}, \mathbf{x}_0) = |\mathbf{x} - \mathbf{x}_0| \int_0^1 ds \sigma_{T,g}(s\mathbf{x} + (1-s)\mathbf{x}_0). \quad (14)$$

Inserting the decomposition (12) into Eq. (11) and using the equation obeyed by $\psi_{gm}^0(\mathbf{x})$, one finds that the equation for $\delta\psi_{gm}(\mathbf{x})$ is

$$(\boldsymbol{\Omega}_m \cdot \nabla + \sigma_{T,g}(\mathbf{x})) \delta\psi_{gm}(\mathbf{x}) = \tilde{Q}_g(\mathbf{x}) + \sum_{g'=1}^G \sigma_{S,gg'}(\mathbf{x}) \delta\phi_{g'}(\mathbf{x}). \quad (15)$$

Notice that this equation has precisely the same form as Eq. (11) for the full flux $\psi_{gm}(\mathbf{x})$, except that the external source Q has been replaced by an effective source $\tilde{Q}$ given by

$$\tilde{Q}_g(\mathbf{x}) = \sum_{g'=1}^G \sigma_{S,gg'}(\mathbf{x}) \phi_{g'}^0(\mathbf{x}) = \sum_{g'=1}^G \sigma_{S,gg'}(\mathbf{x}) \frac{q_{g'} \exp(-\gamma_{g'}(\mathbf{x}, \mathbf{x}_0))}{4\pi |\mathbf{x} - \mathbf{x}_0|^2}. \quad (16)$$

Thus, we have in effect dispersed the point source to a source of “first-scatterers” distributed throughout space. By doing so, it is possible to eliminate some of the severe ray effects associated with the point source. The principal cost in implementing this method lies in the computation of the optical thicknesses $\gamma_g(\mathbf{x}, \mathbf{x}_0)$ for every point in space $\mathbf{x}$.

Finally we must discretize in space. The current version of *CALTRANS* utilizes the well-known diamond difference scheme for this purpose. (As first shown by Ferguson and Greenbaum [3], this scheme can be derived using the Petrov-Galerkin finite element method.) The computational domain is a three-dimensional rectilinear mesh defined by the three coordinate arrays, x_i ($i = 1, \dots, I + 1$), y_j ($j = 1, \dots, J + 1$) and z_k ($k = 1, \dots, K + 1$). On this mesh, we expand the angular flux in linear, piecewise-continuous elements, viz.

$$\psi_{gm}(\mathbf{x}) = \sum_{ijk} \psi_{gm,ijk} \varphi_i(x) \varphi_j(y) \varphi_k(z). \quad (17)$$

Here $\varphi_i(x)$ is the following “teepee” function,

$$\varphi_i(x) = \begin{cases} \frac{x - x_{i-1}}{\Delta x_{i-1}}, & x_{i-1} \leq x \leq x_i, \\ \frac{x_{i+1} - x}{\Delta x_i}, & x_i \leq x \leq x_{i+1}, \\ 0, & \text{otherwise} \end{cases} \quad (18)$$

where $\Delta x_i \equiv x_{i+1} - x_i$, with similar definitions for $\varphi_j(y)$ and $\varphi_k(z)$. Note that the teepee functions are normalized so that $\psi_{gm,ijk} = \psi_{gm}(x_i, y_j, z_k)$, that is, the coefficients $\psi_{gm,ijk}$ give the flux at the *nodes* of the mesh. The equations of the diamond difference scheme are then obtained by replacing $\psi_{gm}(\mathbf{x})$ by expansion (17) in Eq. (11) and by integrating this equation with unit weight over every *zone*. This produces a system of equations of the form,

$$\sum_{\bar{i}\bar{j}\bar{k}} (H_{gm})_{ijk,\bar{i}\bar{j}\bar{k}} \psi_{gm,\bar{i}\bar{j}\bar{k}} = S_{g,ijk}, \quad (19)$$

where $g = 1, \dots, G$, $m = 1, \dots, M$, $i = 1, \dots, I$, $j = 1, \dots, J$, and $k = 1, \dots, K$. The operator H_{gm} consists of two pieces: a streaming term and a removal term. Defining the components of the flux direction via $\boldsymbol{\Omega}_m = (\alpha_m, \beta_m, \gamma_m)$, the streaming term is found to be

$$\begin{aligned} (H_{gm}^{stream} \psi_{gm})_{ijk} &= 4 \int_{x_i}^{x_{i+1}} \frac{dx}{\Delta x_i} \int_{y_j}^{y_{j+1}} \frac{dy}{\Delta y_j} \int_{z_k}^{z_{k+1}} \frac{dz}{\Delta z_k} \boldsymbol{\Omega}_m \cdot \nabla \psi_{gm}(\mathbf{x}) \\ &= \frac{\alpha_m}{\Delta x_i} (\psi_{i+1j+1k+1} - \psi_{ij+1k+1} + \psi_{i+1jk+1} + \psi_{i+1j+1k} \\ &\quad + \psi_{i+1jk} - \psi_{ij+1k} - \psi_{ijk+1} - \psi_{ijk}) \\ &\quad + \frac{\beta_m}{\Delta y_j} (\psi_{i+1j+1k+1} + \psi_{ij+1k+1} - \psi_{i+1jk+1} + \psi_{i+1j+1k} \\ &\quad - \psi_{i+1jk} + \psi_{ij+1k} - \psi_{ijk+1} - \psi_{ijk}) \\ &\quad + \frac{\gamma_m}{\Delta z_k} (\psi_{i+1j+1k+1} + \psi_{ij+1k+1} + \psi_{i+1jk+1} - \psi_{i+1j+1k} \\ &\quad - \psi_{i+1jk} - \psi_{ij+1k} + \psi_{ijk+1} - \psi_{ijk}). \end{aligned} \quad (20)$$

Note that, for each coordinate projection, there are eight terms corresponding to the eight nodes of the zone. The second term of H_{gm} - the removal term - has a similar 8-node structure. One finds

$$(H_{gm}^{removal})_{ijk,\bar{i}\bar{j}\bar{k}} = \frac{1}{2} (\sigma_{T,g})_{ijk,\bar{i}\bar{j}\bar{k}}, \quad (21)$$

where

$$\begin{aligned} (\sigma_{T,g})_{ijk,\bar{i}\bar{j}\bar{k}} &= 8 \int_0^1 ds_x \int_0^1 ds_y \int_0^1 ds_z \sigma_{T,g}(x_i + s_x \Delta x_i, y_j + s_y \Delta y_j, z_k + s_z \Delta z_k) \\ &\quad \times [\delta_{i+1\bar{i}} s_x + \delta_{i\bar{i}} (1 - s_x)] [\delta_{j+1\bar{j}} s_y + \delta_{j\bar{j}} (1 - s_y)] [\delta_{k+1\bar{k}} s_z + \delta_{k\bar{k}} (1 - s_z)], \end{aligned} \quad (22)$$

where δ_{ij} is the Kronecker delta. For each of the 8 terms, the integrand $\sigma_g(\mathbf{x})$ is linearly weighted towards the corresponding node. If the macroscopic cross section does not vary significantly over a zone, or equivalently if the mesh is sufficiently refined, then this matrix operator reduces to

$$(\sigma_{T,g})_{ijk,\bar{i}\bar{j}\bar{k}} \rightarrow \sigma_{T,g}(x_{i+1/2}, y_{j+1/2}, z_{k+1/2}) [\delta_{i+1\bar{i}} + \delta_{i\bar{i}}] [\delta_{j+1\bar{j}} + \delta_{j\bar{j}}] [\delta_{k+1\bar{k}} + \delta_{k\bar{k}}], \quad (23)$$

whose action on $\psi_{gm,\bar{i}\bar{j}\bar{k}}$ simply sums the 8 nodal fluxes associated with the zone.

The source term is constructed from the right-hand-side of Eq. (11) in the same fashion. One finds that

$$S_{g,ijk} = \begin{cases} 4Q_{g,ijk} + \frac{1}{2} \sum_{g'=1}^G \sum_{\bar{i}\bar{j}\bar{k}} (\sigma_{S,gg'})_{ijk,\bar{i}\bar{j}\bar{k}} \phi_{g',\bar{i}\bar{j}\bar{k}}, & \text{fixed source calculation,} \\ \frac{1}{2\kappa} \sum_{g'=1}^G \sum_{\bar{i}\bar{j}\bar{k}} (\sigma_{S,gg'})_{ijk,\bar{i}\bar{j}\bar{k}} \phi_{g',\bar{i}\bar{j}\bar{k}}, & \text{criticality calculation,} \end{cases} \quad (24)$$

where

$$Q_{g,ijk} = \int_0^1 ds_x \int_0^1 ds_y \int_0^1 ds_z Q_g(x_i + s_x \Delta x_i, y_j + s_y \Delta y_j, z_k + s_z \Delta z_k), \quad (25)$$

$$\phi_{g,ijk} = \frac{1}{M} \sum_{m=1}^M \psi_{gm,ijk}, \quad (26)$$

and $(\sigma_{S,gg'})_{ijk,\bar{i}\bar{j}\bar{k}}$ is evaluated analogously to $(\sigma_{T,g})_{ijk,\bar{i}\bar{j}\bar{k}}$ (cf. Eq. (22)).

A comment about the order of accuracy of the diamond difference scheme is in order: If, for every zone, all 8 nodally-weighted cross sections of Eq. (22) are used in solving the discretized transport equation (19) *and* the flux is sufficiently smooth, then the solution will be second-order accurate. In other words, the nodal fluxes $\phi_{g,ijk}$, evaluated by solving Eq. (19), will deviate from solutions of the continuum equation (11) by terms of order $\mathcal{O}(h^2)$, where h is a typical zone width. However, the first condition, requiring the storage and referencing of 8 nodal cross sections per zone, makes rather heavy demands on memory and computational time. Consequently, it is common practice to replace these nodal cross sections with $\sigma_g(x_{i+1/2}, y_{j+1/2}, z_{k+1/2})$ as given by Eq. (23). This is exact for cross sections that are constant across a zone, but in the general case this replacement reduces the accuracy of solutions to first order. *CALTRANS* has coding that will implement the full second-order scheme, but this is untested, and currently defaults to the faster, but less accurate, first-order scheme.

3 Algorithms

In this section, we outline the layout of *CALTRANS* and describe how it solves the system of equations,

$$\sum_{\bar{ijk}} (H_{gm}^{stream} + H_{gm}^{removal})_{ijk, \bar{ijk}} \psi_{gm, \bar{ijk}} = S_{g, ijk},$$

$$g = 1, \dots, G, \quad m = 1, \dots, M, \quad i = 1, \dots, I, \quad j = 1, \dots, J, \quad k = 1, \dots, K, \quad (27)$$

where H_{gm}^{stream} is given by Eq. (20), $H_{gm}^{removal}$ by Eqs. (21) and (22), and S_g by Eqs. (24) - (26).

A coarse-grained flowchart of *CALTRANS* is given in Figure 1. There are roughly four sets of calculations, or *phases*, contained in the code. In the initialization phase, variables are set according to options selected in the input deck. *CALTRANS* interprets this input deck as a namelist-formatted file; the list of allowed keywords, possible values and defaults are given in Table 1 below. These are read in from the file *input* which resides in the same directory as the executable. Note that the current version only implements uniform zoning in each Cartesian direction.

Figure 1. Flowchart of *CALTRANS*

Keyword	Input		Description
	<i>values</i>	<i>default</i>	
nzones	3 integers	5 5 5	number of zones
boxsize	3 reals	5.0 5.0 5.0	size of mesh (in cm)
mmax	integer	1	number of angles = 4*mmax*(mmax+1)
geom	'UNI'	'UNI'	uniform density
	'SPHERE'		uniform sphere
	'NESTED'		nested spheres
	'TOOL'		ANS Neutron Porosity Logging Tool benchmark
eqn	'INHOMO'	'INHOMO'	fixed source calculation
	'HOMO'		criticality calculation
xsecs	'FAKE2'	'FAKE2'	2-group fake cross sections
	'FAKE16'		16-group Pu/Be cross sections
	'OIL21'		21-group cross sections for ANS benchmark
relax	'JACOBI'	'JACOBI'	Jacobi overrelaxation
	'GAUSS'		Gauss-Seidel relaxation
start	'NEW'	'NEW'	new computation
	'RESTAR'		begin with RESTART file
recycle	integer	10	number of cycles between RESTART dumps
iscatter	0	0	original fixed point source(s)
	1		distributed source of first-scatterers
omega	1.0 - 2.0	1.8	Jacobi overrelaxation parameter
eps	< 1	10 ⁻⁴	convergence criterion
porosity	< 1	0.01	porosity of limestone formation in the ANS benchmark

Table 1. Supported keywords and allowed values for CALTRANS input

The next phase - generation - involves the creation of a number of objects:

- Memory (allocated in subroutine *genstore*).
- A computational mesh (*geomct*).
- Arrays associated with the discrete angles Ω_m (*angles*).
- Density arrays corresponding to the various geometries specified by the input parameter *geom*. These are currently “hard-wired” in the subroutine *mats*.
- The macroscopic removal cross sections $\sigma_{T,g}$ (*sigma*). These are evaluated using the material densities and the microscopic removal ($\tau_{T,g}^{(n)}$) and scattering ($\tau_{S,g,g'}^{(n)}$) cross sections read in by the subroutine *stm*.

- The external source Q_g (*fsrc*).

In the parallel version of the code, a call is also made to the routine *partition*, which partitions certain loop indices amongst the available processors (cf. Appendix I).

The computationally intensive portion of the code is embodied in the iteration phase, so called because the equations (27) are solved by *source iteration*. In essence, we make an initial guess for the scalar flux $\phi_g(\mathbf{x})$ and construct the source term $S_g(\mathbf{x})$ using this guess. The operator $H_{gm}^{stream} + H_{gm}^{removal}$ in Eq. (27) is then inverted to determine $\psi_{gm}(\mathbf{x})$, which in turn is used to produce an improved guess for the scalar flux $\phi_g(\mathbf{x})$. The process is then repeated. For fixed source problems, this cycle is repeated until the relative change in the integrated source,

$$S \equiv \sum_{g=1}^G \sum_{ijk} \Delta x_i \Delta y_j \Delta z_k S_{g,ijk}, \quad (28)$$

falls below a tolerance ϵ specified by the user. This iteration loop and its counterpart for the criticality calculations are diagrammed in Figure 2.

The final phase of *CALTRANS* involves edits of the computed fluxes to derive quantities of interest (counting rates, etc.). The code concludes with a call to the routine *finish* which provides an accounting of the CPU times spent in the various portions of the code.

Figure 2. Flowchart of the subroutine *iterate*

4 Parallelization

In our first pass in parallelizing *CALTRANS*, we have focussed on the computationally expensive portion of the program, namely the iteration loop. In the test problems that we shall describe in the next section, we have found that the initialization, generation and edit phases of the computation consumed less than 2% of the total CPU time required for the largest problems. Furthermore, if we ignore future modifications of these portions of the code, this fraction is expected to decrease as the problems increase in size (e.g. in the number of zones, energy groups, etc.).

Examining the code in finer detail, the iteration phase (Figure 2) consists of a series of multiply-nested loops, all candidates for parallelization. To these we shall apply the simple heuristic:

Parallelize only the “outer” indices of the largest loops.

Our concept of “outer” indices is imprecise, but refers to that set of indices which have minimal dependence on each other or on the other indices of the nested loops. As such, these indices typically appear as - or may be recast as - the outer loops of the nested set. In neutronics codes, we are blessed with a plethora of these outer indices, namely those labelling loops over angles, energy groups and materials. By parallelizing these outer loops, we may safely ignore the “inner” loops over spatial indices (or rather, optimize them with vectorization) and thus avoid the intricacies of spatial domain decomposition.

To proceed with our discussion, let us first introduce some notation and simplification: Recall that M , G and N denote the total number of discrete angles, energy groups, and materials, respectively (in *CALTRANS*, these are given by the parameters *nang*, *ngrps*, and *nmats*, respectively). We shall also assume equal numbers of zones in each Cartesian direction (given by *imax*, *jmax*, and *kmax* in the code), and denote this number by Z .

There are three major sets of loops in *CALTRANS*:

- First there is the sweep over angles and the construction of the new scalar flux $\phi_g(\mathbf{x})$. The directed fluxes $\psi_{gm}(\mathbf{x})$ must be solved for every energy group and angle, at every node. Since the source term $S_g(\mathbf{x})$ is given at this stage of the algorithm, the computation of ψ_{gm} is independent of every other group g' and angle m' . Consequently, the “outer” indices in this case are g and m . The computation for this portion of the code scales like GMZ^3 .
- Next there is the construction of the source $S_g(\mathbf{x})$, or more precisely the scattering contribution to this quantity. Unlike the removal cross section σ_{Tg} , the scattering cross sections,

$$\sigma_{S,gg'}(\mathbf{x}) = \sum_{n=1}^N \frac{N_A \rho^{(n)}(\mathbf{x})}{A^{(n)}} \tau_{S,gg'}^{(n)}, \quad (29)$$

are matrices in the group indices and so are re-computed upon each iteration to save memory. Hence we identify the index n over materials and the indices g and g' over energy groups as “outer”, with the whole source computation scaling like G^2NZ^3 .

- The largest of the remaining loops in the iteration phase, including the computation of the integrated source S (Eq. (28)), scale like GZ^3 .

Due to their large size, we have opted to parallelize the first two sets of loops.

Offhand, it would appear optimal to parallelize the $G \cdot M$ loops over angles and groups in the sweep over angles, and the $G^2 \cdot N$ loops in the evaluation of the source. However, for ease of coding, we have opted to parallelize only the M sub-loops over angles in *sweep* and the $G \cdot N$ sub-loops over groups and materials in *source*. This is a justifiable strategy whenever M and/or $G \cdot N$ exceeds the number of available processors.

The control structures required to implement these parallelizations are very simple. Let us first consider the sweep over angles. Every processor executing *CALTRANS* in a parallel job is associated with a unique integer index called me which runs from 0 to $N_{PROC} - 1$, where N_{PROC} is the total number of processors. This identifier is used by the subroutine *partition* to associate a unique range of angles (ang_start , ang_end) to each processor. (For details on initializing me and the subroutine *partition*, see Appendix I.) If $N_{PROC} > M$, then the first M processors are assigned one angle each while the remaining processors remain idle. In its serial version, the relevant code fragment is

```

do m = 1, nang
  call sweep(m,ng)
  do k = 1, kmaxp
    do j = 1, jmaxp
      do i = 1, imaxp
        phi(i,j,k,ng) = phi(i,j,k,ng) + psi(i,j,k)/real(nang)
      end do
    end do
  end do
end do

```

The subroutine *sweep* computes the angular flux $\psi_{gm}(\mathbf{x})$ for a given group g ($= ng$) and angle m . The loop over nodes ($kmaxp = K+1$, etc.) accumulates the contribution of this angular flux to the scalar flux $\phi_g(\mathbf{x})$. In *CALTRANS*, this code fragment is modified to read

```

if(me+1.le.nang) then
  do m = ang_start, ang_end
    call sweep(m,ng)
  end do
end if

```

```

      do k = 1, kmaxp
      do j = 1, jmaxp
      do i = 1, imaxp
        phi(i,j,k,ng) = phi(i,j,k,ng) + psi(i,j,k)/real(nang)
      end do
      end do
      end do
    end do
  endif
  call gdsun(phi(1,1,1,ng),imaxp*jmaxp*kmaxp,work)

```

(We have deleted some inessential details.) The *if-endif* block handles the case that $N_{PROC} > M$, while the range of the index m over angles has been appropriately limited. The routine *gdsun* is a *global reduction operation* which takes the partially-constructed scalar fluxes extant on each processor and sums these quantities over all processors, returning the sum to all processors. The result is the correct value of the scalar flux:

$$\sum_{me=0}^{N_{PROC}-1} \phi_g^{(me)}(\mathbf{x}) = \frac{1}{M} \sum_{me=0}^{N_{PROC}-1} \sum_{m=ang_start(me)}^{ang_end(me)} \psi_{gm}(\mathbf{x}) = \frac{1}{M} \sum_{m=1}^M \psi_{gm}(\mathbf{x}) = \phi_g(\mathbf{x}). \quad (30)$$

The syntax of *gdsun* follows the conventions of the Intel NX/2 message passing system which is available from the MPSC library on the Meiko CS-2 parallel computer. Its Fortran synopsis is

```

SUBROUTINE GDSUM(x, n, work)
  INTEGER n
  DOUBLE PRECISION x(n), work(n)

```

where x is the input/output vector which is being summed, while *work* is a dummy buffer. In addition to performing the summation, this routine also blocks further computing on all the processors until its work is completed. This blocking is desired in our application since the full scalar flux is immediately required on all processors for the construction of the new source estimate. A non-blocking call could cause some processors to pass partially-constructed scalar fluxes $\phi_g^{(me)}(\mathbf{x})$ to the source computation, leading to erroneous results.

The parallelization of the source computation proceeds along similar lines. This time we have a double summation over groups g' ($= kg$) and materials n ($= nm$), as seen in the serial version of the code:

```

  do kg = 1, ngrps
  do nm = 1, nmats
    call source_sweep(nm,kg,ng)
  end do
  end do

```

The routine *source_sweep* constructs, for given groups g ($= ng$) and g' and material n , the sum,

$$\frac{1}{2} \sum_{\bar{i}\bar{j}\bar{k}} \frac{N_{A\tau_{S,gg'}}^{(n)}}{A^{(n)}} (\rho^{(n)})_{ijk, \bar{i}\bar{j}\bar{k}} \phi_{g, \bar{i}\bar{j}\bar{k}},$$

appearing the scattering contribution to the source $S_{g,ijk}$ ($= sors(i,j,k,ng)$), Eq. (24). In the parallel version, we combine the nested loops into a single loop over a “merged” index $ngnm$, which ranges from 1 to $G \cdot N$ ($= ngrps \cdot nmats$), and distribute this index over processors:

```

if(me+1.le.ngrps*nmats) then
  do ngnm = ngnm_start, ngnm_end
    nm = (ngnm-1)/ngrps + 1
    kg = ngnm - ngrps*(nm-1)
    call source_sweep(nm,kg,ng)
  end do
endif
call gdsum(sors(1,1,1,ng),imax*jmax*kmax,work)

```

Let us summarize - in slightly more abstract language - the steps we took to parallelize *CALTRANS*:

- Identify the loops to be parallelized. Let j be the index of one such loop, with $j = 1, \dots, J$.
- Distribute the range of j equally over the available processors, associating a sub-range (j_{min}, j_{max}) with every processor mc . The computation of j_{min} and j_{max} is performed by the subroutine *partition* during the generation phase (cf. Appendix I).
- Restrict the loop to the sub-range and re-construct the full summations using a global reduction operation.
- Add appropriate initializing and/or terminating calls to the message passing library (cf. Appendix I).

Appendix I also contains details on linking the appropriate message passing libraries at compile time and running *CALTRANS* in parallel mode.

CALTRANS is also capable of running on a network of workstations communicating via the Parallel Virtual Machine software package (version 3.2). Our strategy for doing this is to write a PVM implementation of the *gdsum* subroutine. Basic instructions for running *CALTRANS* using PVM 3.2 are given in Appendix I, while Appendix II contains details of the PVM implementation for the interested reader.

5 Test Problems

We have performed timing studies of *CALTRANS* applied to two test problems. In both problems, the material distribution is described by a simple nested-sphere geometry whose parameters are given in Table 2 below. As a test geometry, these numbers are hard-wired into *CALTRANS* and are invoked by setting *geom* = 'NESTED' in the input deck.

Material	Density (g/cm ³)	Inner Radius (cm)	Outer Radius (cm)
Pu	15.8	0.	5.04
Be	1.83	5.04	8.72
Vacuum	0.	8.72	∞

Table 2. Test problem material distribution.

In the first test problem, we place a fixed source (with a total strength of 1 neutron/sec) at the center of the spheres, and run with two energy groups using the following values for the source strength q_g and the microscopic cross sections, τ_T and τ_S :

$$\begin{aligned}
 q_1 &= 1 \text{ sec}^{-1}, \\
 q_2 &= 0, \\
 \frac{N_A \tau_{T,g}^{(n)}}{A^{(n)}} &= 0.2 \text{ cm}^{-1}, \quad g = 1, 2, \quad n = 1, 2, \\
 \frac{N_A \tau_{S,gg'}^{(n)}}{A^{(n)}} &= \begin{pmatrix} 0.1 & 0.0 \\ 0.1 & 0.1 \end{pmatrix}_{gg'} \text{ cm}^{-1}, \quad g, g' = 1, 2, \quad n = 1, 2.
 \end{aligned}$$

The values of τ_T and τ_S used for this problem are completely arbitrary, and are only used to test the code. They are invoked by setting *xsecs* = 'FAKE2' in the input deck. In the second problem, we performed a (pseudo-)criticality calculation, which of course requires realistic cross sections. For this purpose, we have constructed a 16-group cross section dataset called *ptxsecs* derived from the ENDF library. The dataset is read in by the routine *stmread* when *xsecs* is set equal to 'FAKE16' in the input deck.

The parameter sets used for the two test problems are summarized in Table 3. The upper half of these table gives the actual input parameters, while the lower half gives the values of derived quantities. $Z + 1$ gives the number of nodes along each side of the cubic mesh. Input parameters not listed in Table 3 are given their default values as specified in Table 1; in particular, source iteration was performed using Jacobi overrelaxation (with $\omega = 1.8$) until relative changes in the integrated source **S** or the eigenvalue κ fell below a tolerance of $\epsilon = 10^{-4}$.

Problem	Fixed source	Criticality
nzones	9, 19, 39	9, 19
boxsize	18., 18., 18.	18., 18., 18.
mmax	2	1
geom	'NESTED'	'NESTED'
equ	'INHOMO'	'HOMO'
xsecs	'FAKE2'	'FAKE16'
M (= nang)	24	8
G (= ngrps)	2	16
N (= nmats)	2	2
$Z + 1$	10, 20, 40	10, 20

Table 3. Test problem parameters.

Note the load distribution in these two computations: In the first problem, most of the work occurs in the sweep over 24 angles, while the source evaluation loops only over $G \cdot N = 4$ values (modulo an overall factor of G). On the other hand, the source update requires $G \cdot N = 32$ loops in the criticality calculation, while sweeping over only 8 angles.

We have run these test problems on the 4 machines listed in Table 4.

Machine	Compiler/flags	Total CPU time (sec)	
		Fixed source	Criticality
Cray YMP	cf77	25	14
Cray YMP	cf77 -Wf"-o novector",	29	29
IBM RS6000	xlf -O2 -qddim, cc -O2	102	135
SGI Indigo	f77 -O -r8, cc -O	137	150
Meiko CS-2	pgf77 -O2 -r8	73	61

Table 4. Computers used to run the *CALTRANS* test problems.

All times reported in this table correspond to computations performed on a single processor using $(Z+1)^3 = 40^3$ nodes for the fixed source problem and 20^3 nodes for the criticality calculation. Compiler flags are set for optimized code and 64 bit arithmetic. Some machine-dependent optimizations of the *sweep* routine were made; significant improvement on the Cray (20%) and the SGI (40%) is obtained when variables giving the strides on do-loops are replaced by constants. For future reference, we have also listed for the IBM and SGI machines the C-compiler options used to build PVM 3.2 executables. Results from runs on the Cray YMP are given both with and without vectorization. Runs on the other machines were all made without vectorization; hereafter we shall only quote results from unvectorized calculations.

Table 4 yields the interesting result that the *ratio* of CPU times for the two types of calculations differ somewhat from machine to machine. Since all user-specified parameters have been held fixed, this can only be due to the differing speeds with

which each machine executes different types of instruction. In other words, there is no overall scale factor that may be applied in relative comparisons of machine performance. This is also illustrated in Table 5, which separates the execution times not by type of problem but by type of loop.

Machine	T_{sweep}/GMZ^3	T_{source}/G^2NZ^3	T_{serial}/GZ^3	
			Fixed source	Criticality
Cray YMP	1.5 ± 0.3	0.48 ± 0.04	0.5 ± 0.1	0.8 ± 0.1
IBM RS6000	4.5 ± 0.3	2.6 ± 0.1	5.7 ± 0.6	7.1 ± 0.5
SGI Indigo	6.3 ± 0.6	2.6 ± 0.1	7.4 ± 0.4	9.4 ± 0.7
Meiko CS-2	3.0 ± 0.2	0.92 ± 0.1	2.6 ± 0.2	3.5 ± 0.4

Table 5. Measured loop times (in μsec) per iteration scaled by the number of groups G , angles M , materials N and zones Z^3 .

Three measured times are presented here: two of them, T_{sweep} and T_{source} , giving the times required to sweep over angles and construct the source, correspond precisely to the loops parallelized in the previous section. The third time, T_{serial} , is the difference between these times and the total time T required to execute one master loop of the source iteration algorithm (e.g. the loop portrayed in Figure 2): for a single processor, $T_{serial} = T - T_{sweep} - T_{source}$. For this time, we distinguish between the fixed source and criticality calculations, since different sets of computations occur in each case. (We have called it a “serial” time since it is the only portion of the master loop that we do not parallelize; its importance shall be evident in a moment.) From this table, one sees how the CPU times of different sets of instructions fail to scale from machine to machine.

Next we turn to the parallel performance of *CALTRANS*. In this case, we must also consider the contribution T_{comm} of interprocessor communication to the time it takes to execute one master loop of source iteration. In the limit that $M, G \cdot N \gg N_{PROC}$, a formula for this time is

$$T(N_{PROC}) = \frac{T_{sweep} + T_{source}}{N_{PROC}} + T_{serial} + T_{comm}(N_{PROC}), \quad (31)$$

provided we parallelize the sweep over angles and the source update calculation as described in the previous section. The times, T_{sweep} and T_{source} , refer to their values at $N_{PROC} = 1$. This formula also holds for $M, G \cdot N \sim N_{PROC}$ if N_{PROC} divides evenly into M and $G \cdot N$; modifications of Eq. (31) are easily developed for other cases.

Before going on, it is necessary to clarify what we have loosely referred to as the “interprocessor communication” time actually refers the cumulative time it takes to execute the calls to *gdsum*, which we recall is the routine summing its REAL^*8 vector arguments over all processors. Thus, our measurements of T_{comm} are “contaminated” - perhaps dominated - by contributions that have nothing to do with communication, such as the floating-point arithmetic necessary to sum the vectors. Furthermore,

T_{comm} involves not one- but two-way transmission times, since one transmission of a vector from every processor must be followed by a broadcast to every processor of the vector sum. Nevertheless, we have retained the name since it represents the actual cost in time that the program experiences in order to run in parallel mode.

The measurement of T_{comm} is more subtle than one might expect. We have used code fragments of the form,

```

tim_comm = 0.
:
:
call gsync()
t0 = timer()
call gdsum(phi(1,1,1,ng),imaxp*jmaxp*kmaxp,work)
tim_comm = tim_comm + timer() - t0

```

where *timer* is a routine that returns the amount of time (in seconds) that has elapsed since program execution began. Note the call to *gsync*, whose function is to synchronize all processors, that is, to block each processor at this point until all processors have reached it. This synchronization is necessary to get accurate values for T_{comm} ($= \text{tim_comm}$) since without it T_{comm} on some processors, blocked by *gdsum*, would be excessively inflated by time waiting for other processors. (Before we devised this *gsync* fix, we encountered such distortions in T_{comm} on the 3-node Meiko CS-2 prototype, where one processor was significantly faster than the other two.) Like *gdsum*, *gsync* is available in the Meiko CS-2 tagged message passing library. We have also written our own implementation of *gsync* for running on PVM-linked networks (cf. Appendix II).

There is another difficulty in obtaining T_{comm} related to the lack of appropriate timing tools for Fortran applications. The available utilities (*mclock*, *second*, *etime*, and others) typically only measure the CPU time elapsed in a given process on a particular node of the parallel machine. In other words, it is likely that interprocessor communication times are ignored by these utilities. To determine this, we have re-run the test problems using cruder utilities measuring wall-clock time (with a typical resolution of one second), and compared it with our previous timing results. No discrepancies were detected by this test in the case of the Meiko CS-2,¹ but in the case of the SGI workstations there were major differences. Similar tests were not possible for the cluster of IBM RS6000 workstations we ran on (members of LLNL's Open Computing Facility); these machines were not dedicated to our sole use, implying that any wall-clock time we could measure would inevitably reflect contention by other users. Therefore, at this juncture, it is necessary to issue a disclaimer regarding the timing results we report below for the IBM RS6000: The times used for T_{comm} in

¹We have confirmed our Meiko CS-2 timing results with the newly-released utility *elan.clock* which assesses the nanosecond clock on the Elan communications chip. Its resolution, limited by clock drift from processor to processor, is less than one microsecond.

this case are CPU times, obtained using the *mclock* utility, which may or may not be accurate.

To good approximation, we find that T_{comm} is linear in the size of the messages being communicated. There are two calls to *gdsun* for every source update and every group, with approximate message sizes of $8Z^3$ bytes apiece, so that

$$T_{comm}(N_{PROC}) = 2G \left(T_0(N_{PROC}) + \frac{8Z^3}{B(N_{PROC})} \right), \quad (32)$$

where T_0 is the *latency* of communications while B is its *bandwidth* in units of bytes per second. T_0 measures the overhead of initiating and receiving transmissions and of the switching times as messages travel up and down the communications network. In all our test problems, on all machines, we have found the latency to be negligible (e.g. $T_{comm} < 1$ msec on the Meiko) compared to the bandwidth-delimited transmission time. Henceforth we will ignore this contribution to T_{comm} .

Table 6 gives our results for the bandwidth of interprocessor communications, as determined by timing calls to *gdsun*. We emphasize again that this is an *effective* bandwidth, which includes contributions from vector arithmetic in addition to the usual communication bandwidth.

Machine	N_{PROC}	Bandwidth (Mbytes/sec)
IBM RS6000	2	0.64 ± 0.01
	3	0.41 ± 0.02
	4	0.64 ± 0.01
	8	0.31 ± 0.01
SGI Indigo	2	0.16 ± 0.02
	3	0.10 ± 0.01
	4	0.082 ± 0.007
Meiko CS-2	2	6.0 ± 0.3
	3	4.9 ± 0.2
	4	3.8 ± 0.2
	6	3.3 ± 0.1
	8	2.7 ± 0.1
	12	2.4 ± 0.1
	16	2.0 ± 0.1

Table 6. Effective bandwidths for the global reduction operator *gdsun*.

The poor performance of the SGI workstations is immediately evident, and may be due to small bandwidth of this Ethernet-linked network; our results indicate a one-way bandwidth of 0.15 - 0.3 Mbytes/sec, which may be compared to the pure communication bandwidth of 0.8 ± 0.2 Mbytes/sec measured in an independent study [4].

The bandwidth results given in Table 6 systematically decrease with N_{PROC} in a manner that we fit to the analytic forms,

$$\frac{1}{B(N_{PROC})} = \begin{cases} \frac{1}{B_0} + \frac{N_{PROC} - 1}{B_1}, & \text{IBM, SGI} \\ \frac{1}{B_0} + \frac{\log_2(N_{PROC})}{B_1}, & \text{Meiko} \end{cases} \quad (33)$$

where B_0 and B_1 are positive constants. The results of that fit are given in Table 7. In the case of the IBM² and SGI workstations, the linear dependence of B on N_{PROC} is due to the simple “gather/scatter” algorithm we used to sum vectors over the PVM network. In the case of the CS-2, B may derive its dependence on $\log_2(N_{PROC})$ from at least two sources, which are not distinguished by our timing methods. One possibility is that messages traversing the $\log_2(N_{PROC})$ steps from one processor to another experience proportionately lower bandwidth at the higher levels of the network hierarchy due to contention. Another source is the speed of floating point additions performed in *gdsum*; if the sum is executed as a binary tree, or on a similar logical network, the number of steps required is proportional to $\log_2(N_{PROC})$.

Machine	B_0	B_1
IBM RS6000	0.63 ± 0.02	4.2 ± 0.1
SGI Indigo	0.21 ± 0.05	0.42 ± 0.03
Meiko CS-2	24 ± 2	9.2 ± 0.3

Table 7. Fits to the measured bandwidths of Table 6 using Eq. (33).

B_0 and B_1 are given in Mbytes/sec.

Recalling that our measurements reflect two-way transmission times, our Meiko results give a maximal one-way bandwidth of about 18 Mbytes/sec, which may be compared with the 50 Mbytes/sec of *pure* communication bandwidth advertised by the manufacturer. The suitability of the fit (33) to the measured bandwidths on the Meiko CS-2 (from Table 6) is portrayed in Figure 3.

²The IBM bandwidth results, as shown in Table 6, were erratic and didn’t fit the linear form (33). To arrive at the estimates of B_0 and B_1 given in Table 7, we excluded the $N_{PROC} = 4$ bandwidth results from our linear fit.

5.1 Speed-up and scale-up.

The utility of timing loops and extracting bandwidths is that they permit us to scale up to larger problems and to a larger number of processors. In this regard, it is helpful to define the ratios,

$$\begin{aligned}\epsilon_{serial} &= \frac{T_{serial}}{T_{sweep} + T_{source}}, \\ \epsilon_{comm}(N_{PROC}) &= \frac{T_{comm}(N_{PROC})}{T_{sweep} + T_{source}},\end{aligned}\quad (34)$$

both of which measure the work of serially-processed portions of the source-iteration loop relative to its parallel portion. Then the parallel *speed-up* of the loop is given by

$$\frac{T(1)}{T(N_{PROC})} = N_{PROC} \frac{1 + \epsilon_{serial} + \epsilon_{comm}(1)}{1 + N_{PROC} (\epsilon_{serial} + \epsilon_{comm}(N_{PROC}))}, \quad (35)$$

as may be derived from Eq. (31). It is evident how ϵ_{serial} and ϵ_{comm} control the parallel performance of the program; initially one gains by running on more processors, but as N_{PROC} becomes large, the speed-up approaches an asymptotic value,

$$\frac{T(1)}{T(N_{PROC})} \rightarrow \frac{1 + \epsilon_{serial} + \epsilon_{comm}(1)}{\epsilon_{serial} + \epsilon_{comm}(N_{PROC})},$$

provided ϵ_{comm} is weakly dependent on N_{PROC} (e.g. logarithmic). However, if ϵ_{comm} increases sufficiently rapidly with N_{PROC} , then the speed-up will reach a maximum after which it will monotonically decrease as more processors are added.

Using the scaling laws of T_{sweep} , T_{source} , and T_{serial} , and Eq.(32), one may recast the ratios (34) as follows:

$$\begin{aligned}\epsilon_{serial} &\simeq \frac{T_{serial}/GZ^3}{M(T_{sweep}/GMZ^3) + GN(T_{source}/G^2NZ^3)}, \\ \epsilon_{comm}(N_{PROC}) &\simeq \frac{16/B(N_{PROC})}{M(T_{sweep}/GMZ^3) + GN(T_{source}/G^2NZ^3)}.\end{aligned}\quad (36)$$

Recall that the ratios, T_{sweep}/GMZ^3 , T_{source}/G^2NZ^3 , and T_{serial}/GZ^3 , are approximately constant. This has important implications: First, we see that ϵ_{serial} and ϵ_{comm} , and hence the speed-up (35), are approximately independent of the number of zones Z^3 . This is shown in Figures 4 and 5, which displays the predicted CS-2 speed-ups for the fixed source and criticality problems, together with the measured results, for several values of Z .

There is, however, a second and more important consequence of the scaling relations (36) - as the size of the problem increases along dimensions that we parallelize, namely in M and $G \cdot N$, then these ratios decrease, leading to larger speed-ups for a given number of processors. Let us verify this assertion in a fixed source problem with

16 groups, 24 angles and 4 materials modelled on a mesh with 50^3 zones. The predicted time of one source iteration, and the corresponding speed-up, for this problem is given in Table 8 for several values of N_{PROC} .³

Machine	N_{PROC}	$T(N_{PROC})$ (sec)	speed-up
Cray YMP	1	134	-
IBM RS6000	1	611	1.0
	8	184	3.3
	16	211	2.9
Meiko CS-2	1	268	1.0
	16	37	7.3
	64	31	8.5

Table 8. Predicted times per iteration and speed-ups for a fixed source problem with $G = 16$, $M = 24$, $N = 4$ and $Z = 50$.

Here, in the case of the IBM/PVM network, we see how the limitations of communication bandwidth can seriously degrade the speed-up of a parallel application: as N_{PROC} increases from 1 to 8 to 16, ϵ_{comm} increases from 0.09 to 0.19 to 0.30, while ϵ_{serial} remains a respectable 2.1%. (Recall that, due to the timing ambiguities mentioned above, this is a *best case* scenario.) As a result, the speed-up actually decreases as the number of processors increases past $N_{PROC} \simeq 8$.

On the other hand, substantial speed-ups were obtained in the case of the Meiko. The speed-up factors of 7.3 and 8.5 for $N_{PROC} = 16$ and 64 processors, respectively, should be contrasted with the factor of 5.3 (4.3, observed) predicted for the fixed source problem we studied above (with $M = 24$, $G \cdot N = 4$). In general, this ability of a parallel algorithm to speed-up in response to a change in problem size is called its *scalability*, which may be measured by an object called the isoefficiency function [5].

Before leaving this section, let us ask a question that is of vital concern to us in B Division: How does the Meiko CS-2 stack up in performance vis-à-vis the Cray YMP? Table 8 gives us a glimpse at the answer, where we give the expected time per iteration on the Cray YMP, based on its performance on the test problems and as distilled into the loop times given in Table 5. At least in this larger, more realistic, fixed source problem, the CS-2 with 64 processors will outperform the YMP by a factor of 4. It remains to be seen if this relative advantage will persist when computations are vectorized.

³No results are shown for the SGI/PVM case, whose predicted "speed-up" for two processors is expected to be less than one!

6 Concluding remarks

CALTRANS is a code in development, and does not yet have all the features we would want in a three-dimensional neutronics package. We have in mind a number of improvements and/or extensions that would greatly enhance its attractiveness to users. Among these are

- Vectorization.
- Data distribution.
- Full non-uniform zoning of the Cartesian mesh.
- Higher-order moments (e.g. anisotropies) in the scattering matrix.
- More accurate operators H based on expanding the angular flux in *quadratic* finite elements (cf. Ref. [6]).
- Fission source terms.
- Time dependence.
- (n, γ) coupling.
- Reflective boundary conditions.
- An option for diffusion acceleration.
- Extension from single- to multi-block capability.

Despite its early stage of development, it is hoped that this report on the parallel constructs and performance of *CALTRANS* will be of assistance to our colleagues in B Division and elsewhere in the Laboratory.

We are grateful for discussions and correspondence with Eric Brugger, Milo Dorr, Doug East, John Engle and Charles McMillan. This work was supported by Laboratory Directed Research and Development (LDRD) under grant no. 92-SR-009 and by the Department of Energy under Contract No. W-7403-ENG-48.

Appendix I. Parallel Constructs in *CALTRANS*

At present, *CALTRANS* has three different modes of operation:

- Serial mode, using only one processor.
- Parallel mode, using any number of processors on the Meiko CS-2.
- Parallel mode, using any number of workstations and/or mainframe computers linked together in a Parallel Virtual Machine using PVM 3.2 software.

We discuss each in turn.

Serial mode. To run in serial mode, two parameters must be initialized in the subroutine *initlz*:

```
NPROC = 1
me = 0
```

Parallel mode, using the Meiko CS-2. The following calls must be invoked in the subroutine *initlz*:

```
call mpvc_init()
NPROC = numnodes()
me = mynode()
```

In addition, the flags

```
-Bdynamic -L/opt/MEIKOcs2/lib -lmpvc -lew -lelan -laio
```

must appear on the compile command to link the appropriate message passing routines to the executable *caltrans*. The executable is then run using the Meiko *prun* utility. For example, if one wishes to run *caltrans* on 4 processors in the parallel partition p0-32, the appropriate command is

```
prun -n4 -r RMS_PARTITION=p0-32 caltrans
```

Parallel mode, using PVM 3.2. A Parallel Virtual Machine is collection of machines, or *hosts*, that communicate with one another using the daemon process *pvm*. Initially, the user invokes the PVM daemon on a single host and supplies it with a *hostfile* containing the list of machines the user wishes to link together. The PVM daemon uses this hostfile to fire up PVM daemons on all the host machines. These daemons then await applications that will access them for interhost communication. (For further details, consult the PVM manual [7].) *CALTRANS* enrolls in this PVM network by performing the call,

```
call spawn_pvm
```

in the subroutine *initlz*. Upon execution, the subroutine *spawn_pvm* prompts the user for the number of processors N_{PROC} ($= NPROC$), spawns $N_{PROC} - 1$ child processes all running the same executable, assigns each processor an integer identifier mc (with $mc = 0$ identifying the original parent process) and constructs a task identifier array for future interprocessor communication. At the conclusion of *CALTRANS*, in the subroutine *finish*, the call,

```
call exit_pvm
```

exits from the PVM network. Full texts of *spawn_pvm* and *exit_pvm* are given in Appendix II. Execution of a PVM-linked executable, beginning for example from an SGI parent machine, would proceed as follows:

```
pvmd3 hostfile &
./pvm3/bin/SGI/caltrans
```

The executable *caltrans* must reside where the PVM daemon can find it, by default in the subdirectory *./pvm3/bin/\$(ARCH)*, where *\$(ARCH)* denotes the machine architecture. However, one may assign arbitrary pathnames to the executable by appropriately modifying the hostfile.

Another essential component for parallel mode operation, besides the initialization and control structures described above, is the subroutine *partition*, which partitions the outer indices of parallelized loops amongst N_{PROC} processors. As described in section 4, there are two places in the code where this is required:

1. The computation of the angular flux $\psi_{gm}(\mathbf{x})$ that occurs in *sweep* is partitioned in angles. The subroutine *partition* evaluates the first (*ang_start*) and last (*ang_end*) angle indices associated with the current processor (labelled by the integer mc).
2. The updates of the source function occurring in the subroutine *source* are partitioned in both energy groups and material number. These are indexed by the combined counter *ngnm*. The subroutine *partition* evaluates the first (*ngnm_start*) and last (*ngnm_end*) indices associated with the current processor mc .

Special provisions are made for the case that N_{PROC} ($= NPROC$) is greater than the number of angles M or $G \cdot N$ ($= \text{ngrps} \cdot \text{nmats}$). The full text of *partition* is given below. Notice that the subroutine also works in the case of $N_{PROC} = 1$, placing all angles and groups/materials in one processor, $mc = 0$.

```

subroutine partition
include 'caltrans.h'
integer width, excess, local_width

if (nang.gt.NPROC) then
    width = nang / NPROC
    excess = nang - NPROC*width
    if( me .lt. excess ) then
        local_width = width + 1
        ang_start = me*local_width
    else
        local_width = width
        ang_start = me*width + excess
    end if
    ang_start = ang_start + 1
    ang_end = ang_start - 1 + local_width
else
    ang_start = me + 1
    ang_end = me + 1
endif

ngnm = ngrps*nmats
if (ngnm.gt.NPROC) then
    width = ngnm / NPROC
    excess = ngnm - NPROC*width
    if( me .lt. excess ) then
        local_width = width + 1
        ngnm_start = me*local_width
    else
        local_width = width
        ngnm_start = me*width + excess
    endif
    ngnm_start = ngnm_start + 1
    ngnm_end = ngnm_start - 1 + local_width
else
    ngnm_start = me + 1
    ngnm_end = me + 1
endif

return
end

```

Appendix II. PVM 3.2 Implementation

This appendix contains details of the subroutines *spawn_pvm*, *exit_pvm*, *gdsun* and *gsync* that are used to run *CALTRANS* on a Parallel Virtual Machine. These listings are provided with little commentary, save for that provided by their own internal documentation. For synopses and discussion of the PVM library calls, the reader is referred to the PVM manual, Ref. [7].

The subroutine *spawn_pvm* operates in two modes, either as a child or as the parent. If a parent, it spawns off $N_{PROC}-1$ child processes, each running the same executable *caltrans*. Much of the machinery of this routine is devoted to creating and exchanging a task identifier array by which every processor can address another. In particular, the process *me* has a task identifier of *tid(me)*.

```
subroutine spawn_pvm
integer mytid
include 'caltrans.h'
include '../pvm3/include/fpvm3.h'

c  Enroll in pvm
    call pvmfmytid( mytid )

c  Find out if I am parent or child - spawned processes have parents
    call pvmfparent(tid(0))
    if( tid(0) .lt. 0 ) then
        tid(0) = mytid
        me = 0
        print *, 'Give number of processes, NPROC'
        read *, NPROC
        if(NPROC.eq.1) return

c  Start up copies of myself
        call pvmfspawn( 'caltrans', 0, '*', NPROC-1, tid(1), info)

c  Multicast NPROC to children
        call pvmfinit send( 0, info )
        call pvmfpack( INTEGER4, NPROC, 1, 1, info )
        call pvmfmcast( NPROC-1, tid(1), msgtag6, info )

c  Multicast tid array to children
        call pvmfinit send( 0, info )
        call pvmfpack( INTEGER4, tid, NPROC, 1, info )
        call pvmfmcast( NPROC-1, tid(1), msgtag0, info )

    else
```



```

c  Receive NPROC from parent
    call pvmfrecv( tid(0), msgtag6, info )
    call pvmfunpack( INTEGER4, NPROC, 1, 1, info )

c  Receive the tid array and set me
    call pvmfrecv( tid(0), msgtag0, info )
    call pvmfunpack( INTEGER4, tid, NPROC, 1, info )
    do 30 i=1, NPROC-1
        if( mytid .eq. tid(i) ) me = i
30    continue
    endif

    return
end

```

The *tid* array is passed to this routine via the header file *nodes.h*, given below, which is included in the header file *caltrans.h*. Also defined there are identifiers that 'tag' each type of message. While not essential for *CALTRANS*, the use of message tags is good programming practice, serving as an insurance policy against possible mix-ups in communications when *CALTRANS* is linked with other parallel applications.

c Header file: *nodes.h*

```

integer NPROC, MAXNPROC, me, tid
parameter (MAXNPROC = 64)
common /pvm/ NPROC, me, tid(0:MAXNPROC-1)

```

c Message tags for PVM communications

```

integer msgtag0, msgtag1, msgtag2, msgtag3, msgtag4, msgtag6
parameter (msgtag0 = 0) ! multicasting tid to child processes
parameter (msgtag1 = 1) ! gathering data for gdsum
parameter (msgtag2 = 2) ! broadcasting sum from gdsum
parameter (msgtag3 = 3) ! gathering signals for gsync
parameter (msgtag4 = 4) ! broadcasting "Go!" from gsync
parameter (msgtag6 = 6) ! multicasting NPROC to child processes

```

The subroutine *exit_pvm* removes *CALTRANS* from the PVM network. It disengages the application but does not halt the PVM daemons, which continue to run in background.

```

subroutine exit_pvm
include 'caltrans.h'

```

```

include '../pvm3/include/fpvm3.h'

if(NPROC.gt.1) call pvmfexit(info)

return
end

```

Here is our PVM 3.2 implementation of the Meiko global sum operation *gdsum*. Note that it uses a simple gather/scatter algorithm, with the parent ($me = 0$) performing the work of gathering vectors from the children, summing them, and then broadcasting the sum back to the children. The algorithm is inefficient, requiring $O(N_{PROC})$ operations to execute, while a binary sum algorithm could reduce this to $O(\log_2 N_{PROC})$. The PVM routine *pvmfrecv* is a blocking receive, serving to block all processors from further computation until the summed vector has been received.

```

SUBROUTINE gdsum(x,nx,work)
  INTEGER nx
  REAL x(nx), work(nx)
  DATA izero /0/

  include 'nodes.h'
  include '../pvm3/include/fpvm3.h'

  IF (me.eq.0) THEN

    DO ip = 1, NPROC-1
      call pvmfrecv(tid(ip),msgtag1,info)
      call pvmfunpack( REAL8, work, nx, 1, info)
      DO n = 1, nx
        x(n) = x(n) + work(n)
      END DO
    END DO

    call pvmfinit send(0,info)
    call pvmfpack( REAL8, x, nx, 1, info)
    call pvmfmcast(NPROC-1,tid(1),msgtag2, info)

  ELSE

    call pvmfinit send(0,info)
    call pvmfpack( REAL8, x, nx, 1, info)
    call pvmf send(tid(0),msgtag1, info)

    call pvmfrecv(tid(0),msgtag2,info)
  
```

```
        call pvmfunpack( REAL8, x, nx, 1, info)

ENDIF

RETURN
END
```

Finally, we do not provide text of our PVM implementation of *gsync*, as it is nearly identical to *gdsun*. Instead of summing a vector, *gsync* merely passes a token (e.g. an integer) between parent and child. The token itself is unimportant; rather, it is the calls to *pvmfrece* that block all processors until released by a transmission by the parent. This method of synchronization is not perfect - processors are released at time intervals comparable to the latency of communication - but this sufficed for the timing studies reported in Section 5.

References

- [1] K.M. Case and P.F. Zweifel, *Linear Transport Theory* (Addison - Wesley, Reading, Massachusetts, 1967), pp. 17-20.
- [2] E.E. Lewis and W.F. Miller, Jr., *Computational Methods of Neutron Transport* (Wiley - Interscience, New York, 1984).
- [3] A. Greenbaum and J.M. Ferguson, *J. Comp. Phys.* **64**, 97 (1986).
- [4] Eric Brugger, private communication.
- [5] A.Y. Grama, A. Gupta and V. Kumar, *Isoefficiency: Measuring the Scalability of Parallel Algorithms and Architectures*, IEEE Parallel and Distributed Technology (August 1993), p. 12.
- [6] L. Carson and J. Ferguson, *A Comparison of Numerical Schemes for Transport Problems*, Lawrence Livermore National Laboratory Report, in preparation (1994).
- [7] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam, *PVM 3.0 User's Guide and Reference Manual*, Oak Ridge National Laboratory Report ORNL/TM-12187 (1993).

Figure 3. Bandwidth fit on the Meiko CS-2

Figure 4. Speed-up on the Meiko CS-2

Fixed source problem

Figure 5. Speed-up on the Meiko CS-2

Criticality problem

DATE

FILMED

5/24/94

END